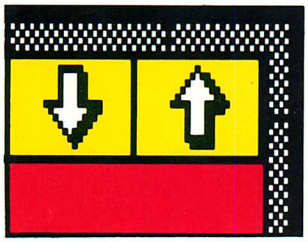
MAY1987 • VOL2 NO2 • TEN DOLLARS

Microsoft® Systems Journal

1

Microsoft Operating System/2: A Foundation for the Next Generation

Microsoft's joint effort with IBM to develop an operating system for Intel 80286- and 80386-based computers resulted in Operating System/2. Using the 286's protected mode, OS/2 not only breaks the 640K memory barrier, it also provides multitasking capability and inter-process communication.



13

OS/2 Windows Presentation Manager: Microsoft Windows on the Future

Included as a standard part of the new Operating System/2, the Windows presentation manager will provide OS/2 with the same advantages that *Microsoft Windows* brings to MS-DOS, such as a standard user interface and a rich independent graphic application environment.

19

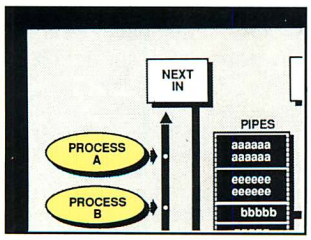
OS/2 DOS Environment: Compatibility And Transition for MS-DOS Programs

An important OS/2 feature is its MS-DOS compatibility environment, a mode that emulates DOS 3.*x* and runs most existing programs. Discussed are the restrictions of the compatibility mode and the considerations in the construction of programs designed to run with both MS-DOS and OS/2.

27

OS/2 Multitasking: Exploiting the Protected Mode of the 286

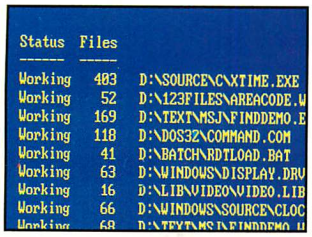
The Microsoft OS/2 scheduler, the dispatcher, and the protected mode of the 286 are the keys to OS/2's multitasking operation. This article considers the system developers view of the fundamental elements of multitasking: processes, threads, and screen groups.



37

OS/2 Inter-Process Communication: Semaphores, Pipes, and Queues

Multitasking operations need fast and reliable communication between multiple processes and threads. To offer the system developer a wide range of facilities to choose from, OS/2 depends on shared memory as well as system semaphores, pipes, and queues for rapid exchange of information.



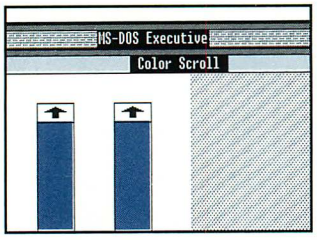
51

A Compleat Guide to Writing Your First OS/2 Program

Discussions of program development for OS/2 introduce a new vocabulary. Dynamic linking or module definition files may at first seem forbidding, but they, like OS/2, are variants of familiar concepts. We offer a sample program complete with multitasking and inter-process communication.

61 **Turning Off the Car to Change Gears: An Interview with Gordon Letwin**

With more than 350,000 lines of code, OS/2 is unquestionably the most complex software that Microsoft has designed. *Microsoft Systems Journal* spoke with Gordon Letwin, the chief architect of OS/2, to learn more about the challenges of designing Operating System/2.



67 **A Simple Windows Application For Custom Color Mixing**

Our regular programming article presents COLORSCR, a deceptively simple program. On the surface, it provides three scroll bars to adjust the red, green, and blue components of GDI color specification. The elegance of the design is its use of child windows to manage the screen area.

18 MICROSOFT UNIVERSITY
34 OS/2 MEMORY MANAGEMENT
47 THE MS OS/2 LAN MANAGER
60 MICROSOFT OS/2 SOFTWARE DEVELOPMENT KIT
76 MICROSOFT LANGUAGES NEWSLETTER
77 ASK DR. BOB

EDITOR'S NOTE

The recently announced IBM Personal Series/2 computers and the first product of Microsoft's joint development agreement with IBM, Microsoft Operating System/2, advance personal computing to new levels of sophistication and improved performance. For the end user, this translates into greater ease of use. For the software developer, however, improvements in technology often mean increasingly complex procedures that bring new challenges and probably a few frustrations as well.

To help the software developer unravel the intricacies of OS/2, this issue of *Microsoft Systems Journal* provides an overview of the new operating system as well as a detailed look at OS/2's basic features—multitasking, inter-process communication, and memory management. And *MSJ* doesn't stop there—the issue continues to explore OS/2 with articles on the MS-DOS compatibility environment, the Windows presentation manager, and an interview with Microsoft's chief architect of OS/2, Gordon Letwin.

What will OS/2 mean to the PC world? Although no one disagrees that the impact will be deep, standards are not made overnight. Because *MSJ* is committed to the concerns and interests of the software developer, we are planning to bring you a great deal more on OS/2, but certainly not to the exclusion of MS-DOS. MS-DOS will continue to be important for a long time to come.

A reminder to DIAL subscribers: Code samples in this and any previous issue are available on-line. Keep an eye on the DIAL Bulletin Board for more information on this.

We've said it before, but it's worth repeating. We want *MSJ* to be *your* journal, *your* forum for exchanging ideas. Write to us with your suggestions for articles, your reactions to this and previous issues.

—Ed.

JONATHAN D. LAZARUS
Editor and Publisher

•

EDITORIAL

TONY RIZZO
Technical Editor

BARRY OWEN
Managing Editor

CHRISTINA G. DYAR
Associate Editor

JOANNE STEINHART
Production Editor

GERALD CARNEY
Staff Editor

DIANA E. PERKEL
Editorial Assistant

•

ART

MICHAEL LONGACRE
Art Director

VALERIE MYERS
Associate Art Director

•

CIRCULATION

WILLIAM B. GRANBERG
Circulation Manager

•

L. PERRIN TOMICH
Assistant to the Publisher

BETSY KAUFER
Office Manager

•

Microsoft Systems Journal (ISSN # 0899-9932) is published bimonthly for a basic U.S. rate of $50.00 per year (single copies, $10.00). International rates provided on request by H. Roark & Associates, Inc., 141 East 44th Street, New York, NY 10017. Application to mail at Second-Class Postage Rates is pending at New York, NY and additional mailing offices. POSTMASTER: Send address changes to Circulation Department, Microsoft Systems Journal, P.O. Box 1903, Marion, OH 43305. Subscription orders should be sent to Microsoft Systems Journal, P.O. Box 1903, Marion, OH 43305. Manuscript submissions and all other correspondence should be addressed to H. Roark & Associates, Suite 804, 141 East 44th Street, NY, NY 10017. You may call direct toll-free: outside Ohio, (800) 533-6625; Ohio residents call, (800) 633-3157.

Copyright © 1987 Microsoft Corporation. All rights reserved; reproduction in part or in whole without permission is prohibited.

The *Microsoft Systems Journal* is a publication of The Microsoft Corporation, 10611 NE 36th Way, Box 97017, Redmond, WA 98073-9717. Officers: William H. Gates, III, Chairman of the Board and Chief Executive Officer; Jon Shirley, President and Chief Operating Officer; Francis J. Gaudette, Treasurer; William Neukom, Secretary.

A Foundation For the Next Generation

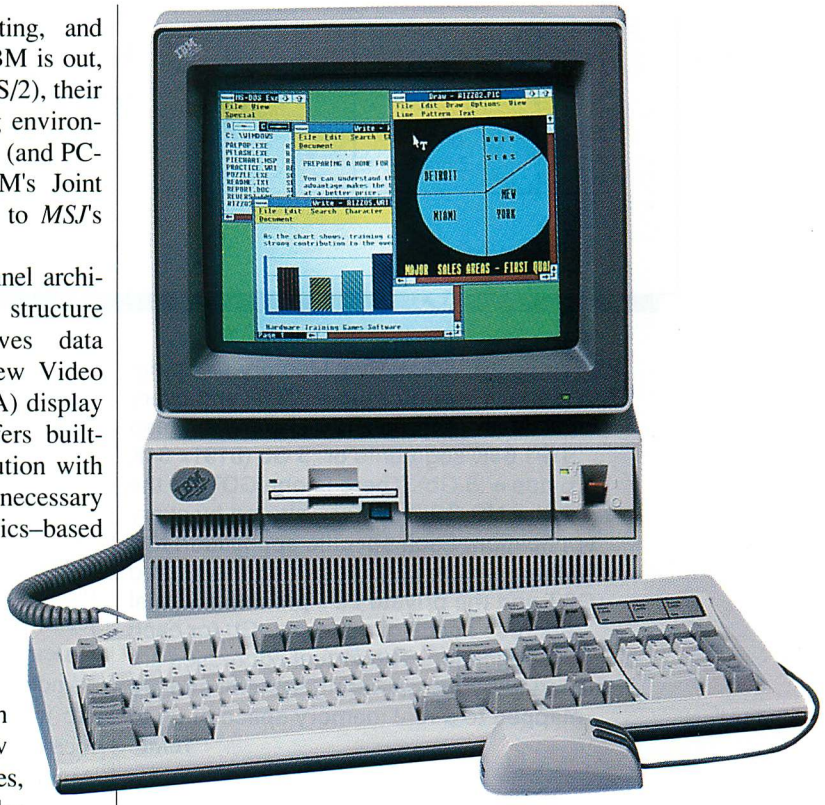
Tony Rizzo

After much waiting, speculating, and hoping, the word from Microsoft and IBM is out, and that word is Operating System/2 (OS/2), their new single-user, multitasking operating environment. OS/2 is the successor to MS-DOS (and PC-DOS) and is the first product of Microsoft and IBM's Joint Development Agreement, which is especially relevant to *MSJ*'s readers.

At the same time that the software announcements were made, IBM launched its next generation of personal computers, the Personal System/2 series. On April 2, 1987, a host of new IBM hardware and software offerings were introduced, but the focus was decidedly on the new IBM machines: an entry-level 8086 PC and new 80286- and 80386-based machines, built mainly around in-house VLSI technology and offering substantial price/performance improvements over the old hardware.

Very briefly, the new systems boast switchless planar (or self-configuring system) boards with on-board video and disk controllers, greatly expanded RAM capabilities, 3-1/2" floppies, and new IBM hard files with a 1-to-1 interleave factor. The new 286 and 386 machines also usher in IBM's 16/32-bit Micro Channel architecture, a new bus structure that greatly improves data throughput, and a new Video Graphics Array (VGA) display subsystem. VGA offers built-in, 640-by-480 resolution with 16 colors, which is necessary for advanced graphics–based software.

For software developers, however, the big news is the new Microsoft OS/2 environment, which will run on the new 286 and 386 machines, as well as on the older PC AT and compatibles.

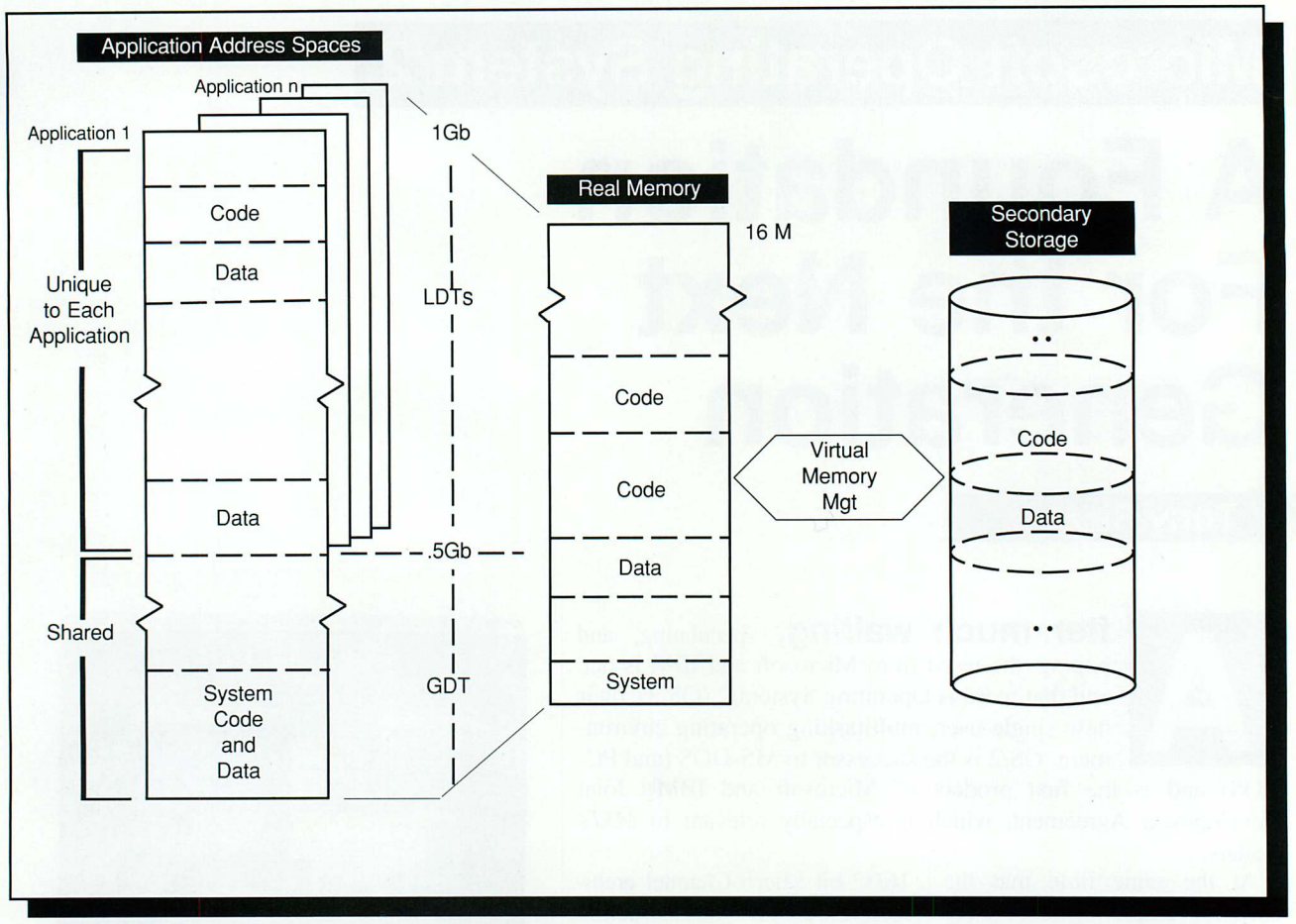


With this issue, Tony Rizzo joins Microsoft Systems Journal *as Technical Editor*

The Connection

In contrast to the current situation in which IBM offers PC-DOS and other manufacturers offer the slightly different MS-DOS, what IBM calls "Operating System/2 Standard

02



▲ **Figure 1 Virtual/Real Memory Mapping. The global descriptor table (GDT) and every local descriptor table (LDT) can each map 8191 64K segments or .5 Gb (8191x 64K bytes = .5 Gb). There is one GDT for the entire MS OS/2 system, shared by every application. Each specific application has its own private LDT. Together, the shared GDT and the private LDT provide a virtual address space of up to one Gb for each application. The virtual addresses are managed by the operating system, and are mapped to a real memory space of up to 16Mb.**

Edition" and what Microsoft calls "Operating System/2" are in fact one and the same operating system.

Perhaps more important is that the Windows presentation manager, the MS OS/2 version of *Microsoft Windows*, is an integral part of the new operating system. Both MS OS/2 and the Windows presentation manager are in compliance with IBM's recently announced Systems Application Architecture (SAA), a set of common specifications created to promote "consistent" applications development across the entire line of IBM System/370, System/3*x*, and personal computer systems.

To date, every version of MS-DOS, including the newly announced Version 3.3, has been a single-process operating system. That is, they can only execute one application at a time. This restriction is a consequence of the 8086/8088 architecture, which is limited to 1Mb of memory and offers no hardware support for protecting memory—any application can access at will any memory location at any time.

Unrestricted memory and device access by software may very well be a boon to those creative programmers who look to squeeze every bit of performance out of the MS-DOS, 8086/8088 environment, but it poses serious problems for those trying to run more than one application at the same time. Furthermore, the memory map of the MS-DOS, 8086/8088 environment leaves

only 640Kb of main memory available for both applications and the operating system.

Though MS-DOS has been continually refined—especially in its file management, device support, background processing (e.g., print spooling), network support, and very rudimentary attempts at multitasking—it has remained fundamentally a single-user, single-application design.

Almost the Promised Land

Without hardware-based protection mechanisms, an operating system would have a very difficult time managing more than one application; preventing applications from accessing conflicting memory locations becomes almost impossible. Intel addressed this problem by including a "protected mode" of operation in the design of its 80286 processor. On account of this, expectations for an enhanced operating system were high when IBM introduced the 286-based PC. The 286 processor's protected mode of operation, its ability to address up to 16Mb of memory, and the fact that the 286 also provided a "real" mode fully compatible with the 8086/8088 suggested the potential for a more powerful operating system.

Unfortunately, the 286 was designed before the importance of the IBM PC (8088) standard was established. As a result, the 286 cannot simultaneously support both real and protected modes—a major drawback. Adding to the problem, the 286 real mode has every deficiency of the 8086 environment: the 640Kb barrier, no memory protection, and, as a result, no multiple-application operation.

Because of these 286 shortcomings, it was not feasible to build a protected-mode operating system with MS-DOS compatibility before the release of the PC AT. So, rather than take a half-step, Microsoft and IBM decided to invest in a new, extensive development effort that would focus on a protected-mode operating system (see "Turning Off the Car to Change Gears: An Interview with Gordon Letwin," page 61). The new operating system would take advantage of the hardware, break the 640Kb memory barrier, allow a multi-application, multitasking environment, and still permit total compatibility with MS-DOS applications.

The result of this effort is MS OS/2.

The Real Thing

MS OS/2 is a priority-based, pre-emptive, multitasking environment which includes a new application interface that isolates applications from low-level hardware differences between machines running MS OS/2. OS/2 utilizes the protected mode of the 286 chip to allocate an exclusive address space to each application. OS/2 controls access to memory, so that different applications cannot corrupt each other's data and program areas.

In protected mode, the 286 uses the contents of the segment register to access a local descriptor table (LDT) containing information about each application. The contents of the segment register serve as an index into the descriptor table. The operating system also accesses a global descriptor table (GDT), which contains information on code and data that is shared by all executing applications (see **Figure 1**). Both of

MICROSOFT AND IBM DECIDED TO INVEST IN A NEW, EXTENSIVE DEVELOPMENT EFFORT THAT WOULD FOCUS ON A PROTECTED-MODE OPERATING SYSTEM. THE NEW OPERATING SYSTEM WOULD TAKE ADVANTAGE OF THE HARDWARE, BREAK THE 640KB MEMORY BARRIER, ALLOW A MULTI-APPLICATION, MULTITASKING ENVIRONMENT, AND STILL PERMIT TOTAL COMPATIBILITY WITH EXISTING MS-DOS APPLICATIONS.

Evolution and History of MS-DOS®

IN AUGUST 1981 Microsoft introduced MS-DOS, Version 1.0, a simple operating system providing support for a new IBM machine and a fledgling industry. Over the last six years the machine and the industry have grown up, and MS-DOS has evolved into MS OS/2, a very sophisticated, multitasking operating environment.

Presented here is a brief but complete history of MS-DOS. It is hoped that the reader will find it entertaining, nostalgic, and perhaps revealing. How many readers know, for example, that the original design of the File Allocation Table (FAT) can be traced to Mr. Gates himself?

1974: Intel introduces the 8 bit 8080 processor.

January 1975: MITS introduces the $400 Altair computer; it has no keyboard, no monitor, no disk, and no operating system.

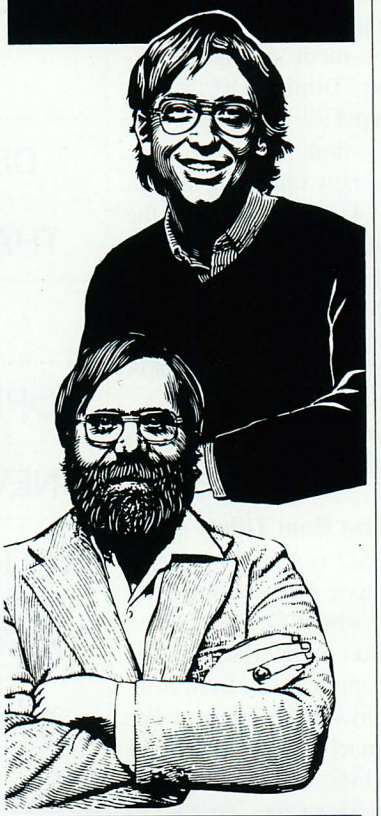
February 1975: Paul Allen and Bill Gates develop and sell their own version of BASIC to MITS for the Altair.

February 1976: Paul Allen, now working for MITS, asks Bill Gates to write a disk-based version of BASIC for the Altair.

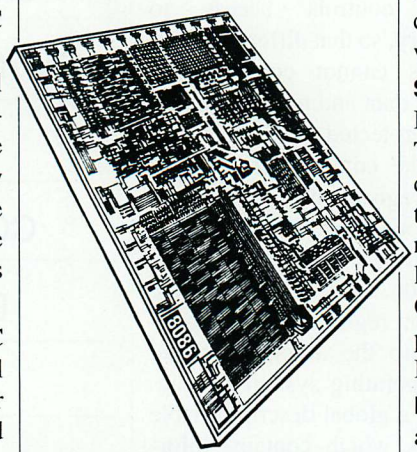
Bill Gates creates a working model of his disk BASIC in 10 days. He designs a disk layout and file structure based on a centralized File Directory and File Allocation Table (FAT). He also includes a rudimentary set of file services in the disk BASIC he is developing.

1976–1978: Microsoft Disk BASIC is ported to all major 8 bit personal computers. An assembler and linker are developed for 8080- and Z80- based systems.

April 1978: Intel announces the 8086, a 16 bit processor.



FEBRUARY 1976



APRIL 1978

January 1979: Tim Paterson of Seattle Computer Products begins work on a plug-in 8086 processor card to bring the power of the 8086 to the S-100 bus.

June 1979: Microsoft and Tim Paterson show Microsoft's BASIC running on Paterson's 8086 card at the National Computer Conference in New York.

April 1980: Delays hold up the delivery of CP/M 86. Tim Paterson decides to write his own "Quick and Dirty" OS, which becomes known as 86-DOS. He incorporates the FAT structure first designed by Bill Gates for Disk-BASIC, and some features and techniques underlying MS-DOS.

August 1980: IBM takes its first step towards producing the IBM PC, planning to use readily available, off-the-shelf 8 bit hardware. IBM visits Microsoft, asking if Microsoft can write a ROM-based BASIC for the computer IBM is developing.

Microsoft suggests that IBM consider the 16 bit architecture. IBM's "Project Chess" goes on to become the 8088 (8086-based) IBM PC.

The first working version of 86-DOS runs on Tim Paterson's 8086 card. This is essentially the birth of what will become known as MS-DOS.

September 1980: IBM asks Microsoft to provide COBOL, FORTRAN and Pascal for their personal computer. Microsoft suggests to IBM that an operating system would be necessary to develop the additional languages.

October 1980: Microsoft submits a proposal to IBM that includes MS-DOS.

November 1980: The proposal is accepted by IBM. A prototype machine arrives at Microsoft and a small DOS team begins a concentrated period of work.

Illustrations by Jerry Breen

February 1981: 86-DOS runs on the prototype for the first time. Over the next half year the OS is refined and becomes MS-DOS, Version 1.0.

August 1981: IBM introduces the IBM PC, and announces three operating systems: MS-DOS, CP/M 86, and the P System. For several months MS-DOS is the only OS available. It is also priced substantially lower than CP/M.

MS-DOS incorporates a number of advanced features:

- Greater device independence.
- Greater data integrity.
- Simplified handling of peripheral devices.
- A replaceable COMMAND.COM processor.

February–April 1982: Microsoft introduces for the MS-DOS environment:

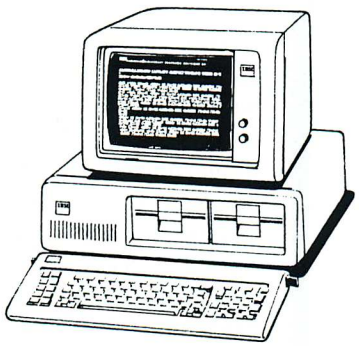
- The Macro Assembler.
- The FORTRAN Compiler.
- The COBOL Compiler.

June 1982: MS-DOS, Version 1.1, is announced, providing support for double-sided, eight sector diskettes on the IBM PC.

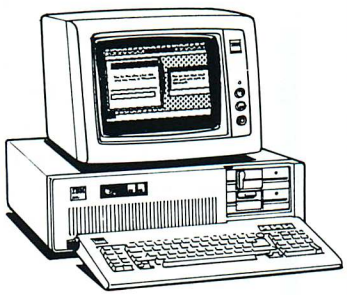
March 1983: IBM announces the PC XT.

Microsoft announces the release of MS-DOS, Version 2.0, which includes:

- Supports for hard disks including a XENIX®-like hierarchical directory structure.
- File Handles, allowing programs to reference files anywhere they have been loaded.
- The ability to redirect the I/O of a program to any file or device, and implements pipes and filters.
- Support for nine-sectored diskettes, increasing storage to 360 Kb.
- Provision for installable device drivers, promoting logical device independence.
- Background printing through PRINT.COM.



AUGUST 1981



AUGUST 1984

APRIL 1987

- ANSI.SYS device driver, providing support for serial "stream" processing to give cursor positioning and color control information to the monitor.

March 1984: IBM introduces the PCjr with half-height disk drives, and MS-DOS, Version 2.11, is introduced, providing support for it.

August 1984: IBM announces the 80286 based PC AT, with a 20 Mb hard disk and 1.2 megabyte high density diskette drive.

Microsoft introduces MS-DOS, Version 3.0, which includes:

- A rewritten MS-DOS kernel and a new standard set of I/O calls.
- ISO Open System Interconnect (ISO) based model for networking.
- Network redirector and file sharing support for the IBM PC Network Adapter Card.

November 1984: Microsoft introduces MS-DOS, Version 3.1, and Microsoft Networks (MS-Net) including:

- Redirector and file sharing services for non-IBM network cards.
- Transport and server functions to all MS-DOS systems.
- Network spooled printing.
- The basic structure for Installable File Systems.

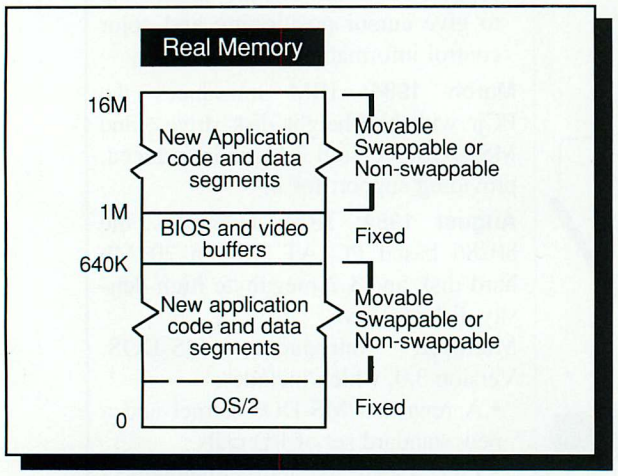
June 1985: Microsoft, Intel, and Lotus establish the Lotus-Intel-Microsoft Extended Memory Specification, allowing programs to access memory beyond the MS-DOS limit of 640 Kb of RAM.

August 1985: IBM and Microsoft enter into a Joint Development Agreement.

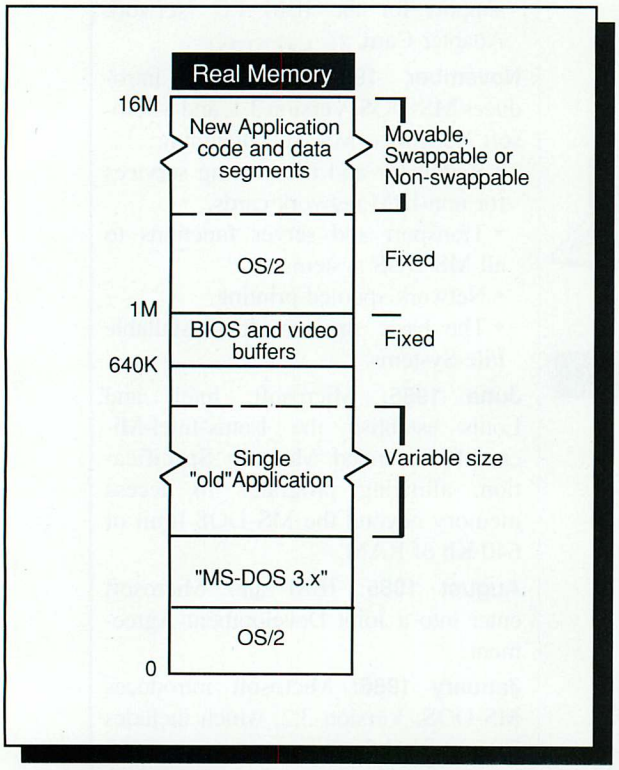
January 1986: Microsoft introduces MS-DOS, Version 3.2, which includes support for 3.5" diskettes.

April 1987: IBM announces the Personal System/2 series of computers.

Microsoft announces MS OS/2, with an integrated Windows presentation manager, MS-DOS, Version 3.3, *Windows* 2.0, and a new LAN Manager. ❑



▲ **Figure 2** Protected Mode Memory Map



▲ **Figure 3** Compatibility Mode Memory Map

these tables are exclusively managed by the operating system, and together they allow up to the entire 16Mb of real memory to be mapped to up to 1 gigabyte of virtual memory.

Also, application programs are normally prevented from writing directly to the underlying hardware. The 286 architecture provides for discrete I/O privilege levels (IOPL) to be assigned to applications; this capability enables the operating system to control access to hardware. Although it is possible for an application to obtain hardware control if it can gain access to an appropriate privilege level, the operating system generally does not explicitly make that privilege available to an application. Protected mode and IOPL are the basic, 286 hardware–based underpinnings of MS OS/2 (for a detailed discussion of privilege levels and how OS/2 performs multitasking, see "OS/2 Multitasking: Exploiting the Protected Mode of the 286," page 27).

Memory Management

In protected mode OS/2 can simultaneously run many applications, any one of which may cause the amount of physical memory available to the system to be exceeded. When an application's memory requirements exceed real memory (referred to as "memory overcommit"), OS/2 will swap memory segments to disk as required (see **Figure 3**). OS/2 dynamically expands and contracts allocated memory and collects fragmented memory allocations (known as "memory compaction"), which provides efficient control of available memory.

Microsoft has gone to great lengths to protect the enormous investment in pre–MS OS/2 software. To achieve this goal, OS/2 provides a compatibility mode in which real-mode MS-DOS applications can run (see **Figure 3**). The compatibility mode offers an 8086 environment in which most MS-DOS programs will run. However, a small number of programs, especially those that directly access hardware or with timing dependence, won't run in compatibility mode.

Compatibility mode allows for one MS-DOS program to execute in the foreground; background, light protected-mode applications are suspended when in this mode. A program executing in the compatibility mode can only run in the foreground—such a program is suspended if one of the background protected-mode applications is selected for foreground operation.

Command Processors

Microsoft OS/2 comes with two command processors, COMMAND.COM and CMD.EXE. The former is the real-mode command processor, similar to the MS-DOS command processor, with its familiar initialization, resident, and transient parts. As in MS-DOS, COMMAND.COM is used to start programs when in compatibility mode.

CMD.EXE is the MS OS/2 command processor, which offers an extended command syntax. Most important, however, is that CMD.EXE is both shareable and swappable. This means that multiple copies of CMD.EXE can be executed, but only one copy of CMD.EXE is actually kept in memory, and that copy is shared by all protected-mode ap-

plications. The only additional memory used is that which is unique to any given application.

Presentation Manager

OS/2 consists of two fundamental, interrelated parts. The first is the kernel, which provides for the major internal workings of the operating system. Integrated into the MS OS/2 system is the Windows presentation manager. Like *Microsoft Windows* for MS-DOS, it is a consistent, graphic user interface and a device-independent applications environment.

The MS OS/2 Windows presentation manager is derived from the current *Microsoft Windows*, Version 1.03, software. The MS OS/2 Windows presentation manager, however, has been significantly enhanced; it has been designed to use multiple overlapping windows and to conform to the Common User Access component of IBM's SAA.

The key word is *consistency*. Microsoft will soon release *Windows*, Version 2.0, for the MS-DOS environment with the same user interface as the Windows presentation manager. The visual interface will be consistent in both MS OS/2 and MS-DOS. Also, because the Windows presentation manager is consistent with SAA, this new standard is expected to eventually find its way into a large number of computer systems, from micros to mainframes.

For the end user, the Windows presentation manager offers

- the ability to run graphics-oriented programs
- a simple and consistent user interface
- access to the session manager
- enhanced mouse and keyboard control

And, for the software developer, the Windows presentation manager provides

- access to the MS OS/2 multitasking and memory management kernel functions
- access to intertask communication and data interchange facilities
- a device-independent output architecture
- a rich and sophisticated graphics library

The Session Manager

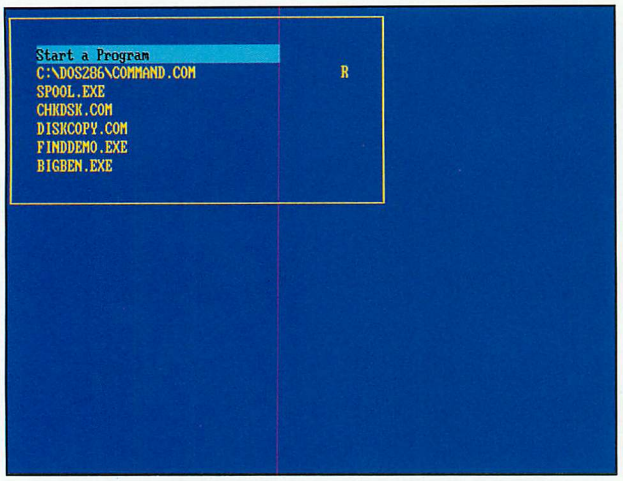
The new *Windows* environment gives the user full control over the operating environment. The windows environment accomplishes this by offering users a session manager, which allows users to control the status of concurrently running applications, as well as that of any application running in compatibility mode. The user requests access to the session manager with the SysReq key, and the session manager responds with a list of active applications (see **Figure 4**). The user can then bring any process to the foreground, start any number of new processes, or terminate any number of existing processes. There is a lot more to the session manager, however, than meets the eye.

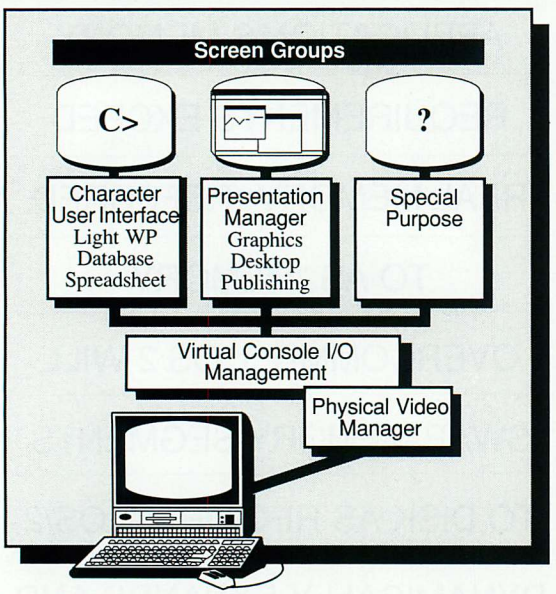
The New Frontier

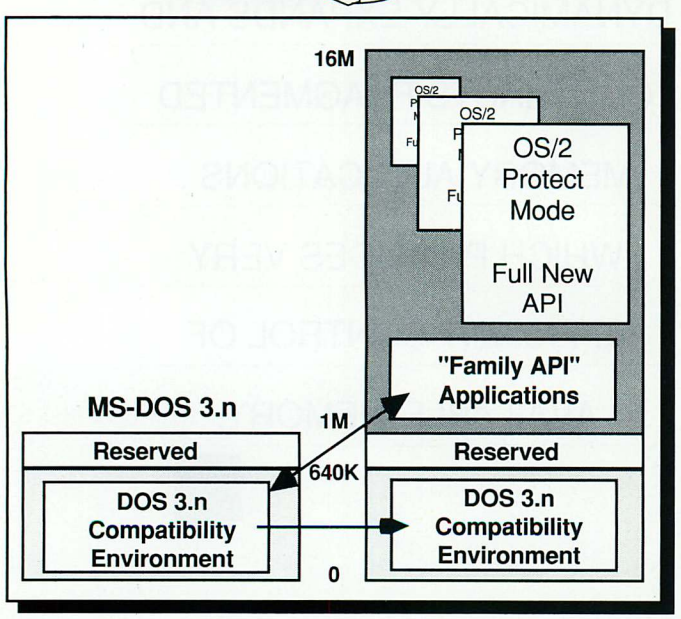
When the session manager is active, it switches between different screen groups. To understand screen groups we need to take a close, if somewhat brief, look at the activity underlying executing applications. MS OS/2 applications are made up of one or more processes, which are in turn made up of

07

IN PROTECTED MODE MICROSOFT OS/2 CAN SIMULTANEOUSLY RUN MANY APPLICATIONS. WHEN AN APPLICATION'S MEMORY REQUIREMENTS EXCEED REAL MEMORY (REFERRED TO AS "MEMORY OVERCOMMIT"), OS/2 WILL SWAP MEMORY SEGMENTS TO DISK AS REQUIRED. OS/2 DYNAMICALLY EXPANDS AND CONTRACTS FRAGMENTED MEMORY ALLOCATIONS, WHICH PROVIDES VERY EFFICIENT CONTROL OF AVAILABLE MEMORY.







one or more threads. A thread is an "atomic unit," a unique entity that the MS OS/2 scheduler can allocate CPU cycles to, that is, it is actually executable by the CPU. A process can have any number of executable threads, each of which can be in one of three states: blocked, ready to execute, or executing.

Processes are, in turn, assigned to screen groups. Screen groups are collections of processes that write to a common virtual display and receive input from a virtual keyboard. As an example, a screen group might be a collection of related windows, each of which is running its own process and each of which was started inside a parent window (screen).

The screen group writes to a virtual display and reads from a virtual keyboard because it is possible to execute any number of screen groups, each with its own screen and keyboard needs. Since there is only one physical screen and keyboard, there must be some system to prevent applications from conflicting with each other. MS OS/2's solution maps each screen group to its own unique virtual screen, eliminating any possible conflicts.

When a user requests access to an active application, the session manager maps that application's screen group to the physical screen—more commonly known as the foreground. MS OS/2 manages the mapping of virtual screens to the physical screen with the session manager (see **Figure 5**). The user has complete control over which applications MS OS/2 will bring to the foreground, and over initiation or termination of new applications (and their associated screen groups).

There is a lot going on here. Obviously, OS/2 has provided the user with a great deal of flexibility. But what has the software developer gained?

▲ Figure 4 (top) The Session Manager

▲ Figure 5 (center) OS/2 maps multiple virtual I/O to the physical screen and keyboard.

▲ Figure 6 (left) FAPI assures the software developer that an application will run under both MS-DOS 3.*x* and MS OS/2.

API and FAPI

MS OS/2 offers a rich application program interface that allows software developers to fully exploit the resources of the operating system. For example, the Windows presentation manager provides a large set of sophisticated graphics functions, giving the software developer the building blocks necessary to create very sophisticated screen images.

MS OS/2 provides a substantial set of CALL-based functions—189 of them to be exact—to assist in the development of programs that will run in a multitasking environment. These functions, which handle file and device management, memory management, and task management, are known as the kernel, or operating system Application Programming Interface (API).

The API includes facilities to write device drivers, known as the Device Driver Interface. Device drivers handle device initialization, transfer of data to and from devices, and device

error control, and they provide access to low-level BIOS functions. High-level languages such as C can call these functions directly. These callable functions completely eliminate the need to depend on interrupts (for example, Interrupt 21 function calls) and the need to write directly to hardware.

API also provides functions to deal with the complexities of inter-process communication (IPC). Processes and threads communicate through semaphores, queues, pipes, and shared memory (see "OS/2 Inter-Process Communication: Semaphores, Pipes, and Queues," page 37). API's functions make such complex programming significantly easier to write.

Microsoft will support all of these functions as a standard. There will be a substantial improvement in the ability to port software to future releases of MS OS/2, as well as an enhanced ability for the products of different software developers to interact with each other. Microsoft suggests that applications conforming to the API standard will run within the MS OS/2 environment and future releases of OS/2. This creates an opportunity to standardize coding practices throughout the entire PC environment.

Realistically, OS/2 won't become as pervasive as MS-DOS for some time yet. It will most likely be at least 3 years before 286- and 386-based machines take over the major share of the PC market. MS-DOS, Version 3.3, will remain with us for the foreseeable future, so developers will want the ability to write software that can move between MS-DOS and MS OS/2.

This need, however, has al-

CMD.EXE—OS/2 Command Processor

09

OS/2 provides the user with a new command processor that extends the functionality of the command language and provides enhanced batch file support.

Command Line Operators

The following shows both the old operators as well as the new, from highest to lowest precedence.

command.com	cmd.exe	Function
	^	**Lexical Escape Charactor**
	()	**Command Grouper**
<	<	**I/O Redirector -+ Same**
>	>	**I/O Redirector |->Precedence**
>>	>>	**I/O Redirector -+ Level**
|	|	**Pipe Operator**
	&&	**And Operator**
	||	**Or Operator**
	&	**Command Separator**

The operators common to both environments work the same way they previously did.

The Lexical Escape Character allows the precedence operators to be used as regular charactors. Refer to the following,

1. echo go home > kid
2. echo go home ^> kid

In the first case a file called kid is created with the line "go home" as its contents. In the second, "go home > kid" is echoed to the screen.

The Command Grouper works in exactly the same way parentheses work in equations, to establish explicit precedence. Refer to the following,

directory a: contains the files temp1, temp2 and temp3
temp1 contains "a"
temp2 contains "b"
temp3 contains "c"

1. dir a:*.* >> temp1 & (temp1 >> temp2 & temp2 >> temp3)
2. (dir a:*.* >> temp1 & temp1 >> temp2) & temp2 >> temp3

After execution of the first command line temp1, temp2 and temp3 will look as follows

temp1 contains a,temp1,temp2,temp3
temp2 contains b,a
temp3 contains c,b,a

If the second line had been executed instead, temp1, temp2 and temp3 would look as follows

temp1 contains a,temp1,temp2,temp3
temp2 contains b,a,temp1,temp2,temp3
temp3 contains c,b,a,temp1,temp2,temp3

The parentheses determine the order of execution. Additionally, lower precedence operators can be executed prior to those of higher precedence.

The Pipe and Redirection operators work in the same way they previously did.

The And operator performs the command to its right if and only if the command to its left successfully executes. For example, in

dir exist.txt && copy exist.txt lpt1

if the DIR command is successful, then exist.txt is printed. If DIR was not successful the command on the right will not be executed.

CONTINUED

CMD.EXE—OS/2 Command Processor

CONTINUED

The Or operator performs one of two commands depending on the success or failure of the first command given. For example, in the following

more < document.txt || more < document.doc

more will either output the contents of document.txt and then exit, or, if document.txt doesn't exist, will output the contents of document.doc.

The Command Separator allows more than one command to be typed on a command line. For example,

dir a:*.* > temp & dir b:*.* >> temp

places the directory listing of the root directory of a: in a file called temp (which is either overwritten if it exists or is created if it doesn't exist) and then immediately executes the next command, which appends the contents of the root directory of b: to temp.

BATCH FILE ENHANCEMENTS

OS/2 supports all of the old batch file commands and adds four new ones. They are

CALL
EXTPROC
SETLOCAL
ENDLOCAL

The Call command finally provides a means for users to chain together batch files so that when a called batch file ends it returns control to the calling batch file.

Extproc allows a user to run batch files designed for command processors other than CMD.EXE. The command is issued as the first line of the batch file and defines the alternate command processor.

Setlocal allows the user to define local drive, directory and environment variables for the current batch file. When the command is issued it saves the existing drive, directory and environment information and replaces it with the user specified information.

In conjunction with the setlocal command, the endlocal command restores the drive, directory and environment information previously saved through setlocal, and allows additional setlocal calls.

If a setlocal call is issued and the batch file issuing the setlocal call terminates without issuing an endlocal call, the operating system restores the original drive, directory and environment information, insuring that the state previously known to CMD.EXE still exists.

ready been anticipated and resolved; Microsoft guarantees to support a subset of the API function set that will execute in every available mode of the two operating environments. Dubbed the Family Application Programming Interface (FAPI), this set of 91 functions—25 of them are somewhat restricted—will permit programs to execute under MS-DOS 3.3, within the compatibility mode of MS OS/2, and as an actual, although limited, MS OS/2 application (see **Figure 6**).

API/FAPI represents a unique opportunity for software developers to create applications that can deal with current limitations while incorporating future advances, thus providing users to run state-of-the-art applications on older machines.

Software Assistance

By August, Microsoft plans to release a Software Development Kit (SDK) that will contain the full specifications of MS OS/2—as well as a beta-release version of the operating system kernel. The toolkit will come with C Version 4.5, MASM Version 4.5, and the CodeView debugger.

The new compiler and assembler will be able to generate code to be used with the new Microsoft Dynamic Linker. The linker will allow code to be linked with FAR calls, creating external references to various libraries. These external references eliminate the need to link library routines into each object (or load) module. The Dynamic Link Editor services program needs by loading library routines only when necessary.

The LAN Manager

The LAN Manager will play an important role in the MS OS/2 environment and can be fully integrated into MS OS/2. It promises an exciting new applications environment in which different applications, such as spreadsheets, word processors, and database systems, will be able to communicate with each other not only application-to-application (e.g., spreadsheet to word processor) within one processor, but also among processors connected to

a network. The LAN API component allows developers to take full advantage of the capabilities of the LAN Manager so as to build truly distributed applications. It will also include site-to-site electronic-mail capabilities and resource sharing. (For more information on the LAN API, see "The MS OS/2 Lan Manager," page 47) .

Help from the 286

The 386 chip provides much faster speed, 1 gigabyte of addressable real memory, and a huge virtual address space—64 terabytes! It is also capable of running "virtual" 8086 machines, obviating the need for a compatibility mode. Users can run multiple MS-DOS sessions within the multitasking environment.

Microsoft estimates that only 10 percent of the MS OS/2 code will have to be modified or rewritten, primarily in the memory management component, to utilize the 386. This code modification will be completely transparent to applications written under the MS OS/2 API on the 286 chip; MS OS/2 has been standardized in this respect. With the next release of MS OS/2, software developers may gain access to additional tools, but they won't have to rewrite any code because of changes made to the existing API. This means, of course, that software developers can begin now to create on the 286 applications that can easily be ported to the 386—a substantial development advantage.

A New Philosophy

MS OS/2 is obviously a greatly improved and sophisticated piece of software. Software development under this new environment will require a considerable effort to understand just how all of the pieces fit together. It will take some time before developers can claim mastery of all the tools currently available.

There will also be a shift in emphasis from programming

System Configuration (CONFIG.SYS)

CONFIG.SYS is the file that contains the commands used to configure the system. Only a single CONFIG.SYS file is needed to configure the system for both real-mode and protected-mode operations.

During system initialization, OS/2 opens and reads the CONFIG.SYS file in the root directory of the drive from which it was started and interprets the commands within the file. If the file is not found, OS/2 assigns default values for the configuration commands.

The following list summarizes the configuration commands for OS/2:

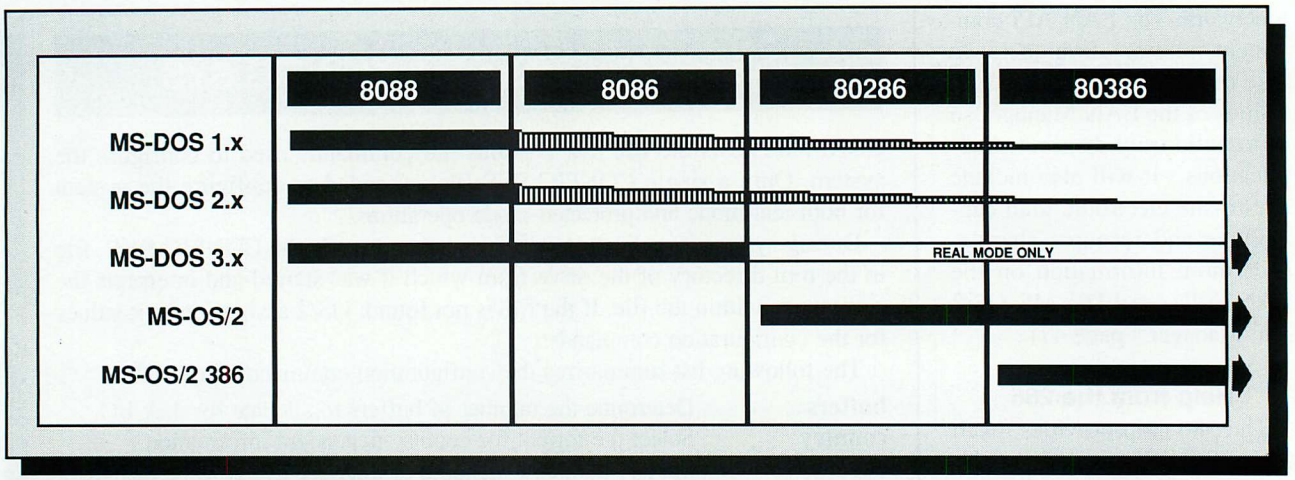
buffers	Determine the number of buffers to allocate for disk I/O.
country	Select the format for country-dependent information.
device	Specify path and filename of a device driver to be installed.
iopl	Specify whether I/O privilege is to be granted or not.
maxwait	Set the time limit for calculating CPU starvation.
memman	Select memory management options.
priority	Disable dynamic priority variation in scheduling regular class processes.
protectonly	Select the modes of operation.
protshell	Specify path and filename of the OS/2 top-level command processor.
run	Start a system or daemon process during system initialization.
swappath	Specify the location of the swap file.
timeslice	Set the time slice values for process scheduling.
threads	Set the maximum number of threads in the system.
trace	Select system trace.
tracebuf	Set the system trace buffer size.

The following commands are ignored:

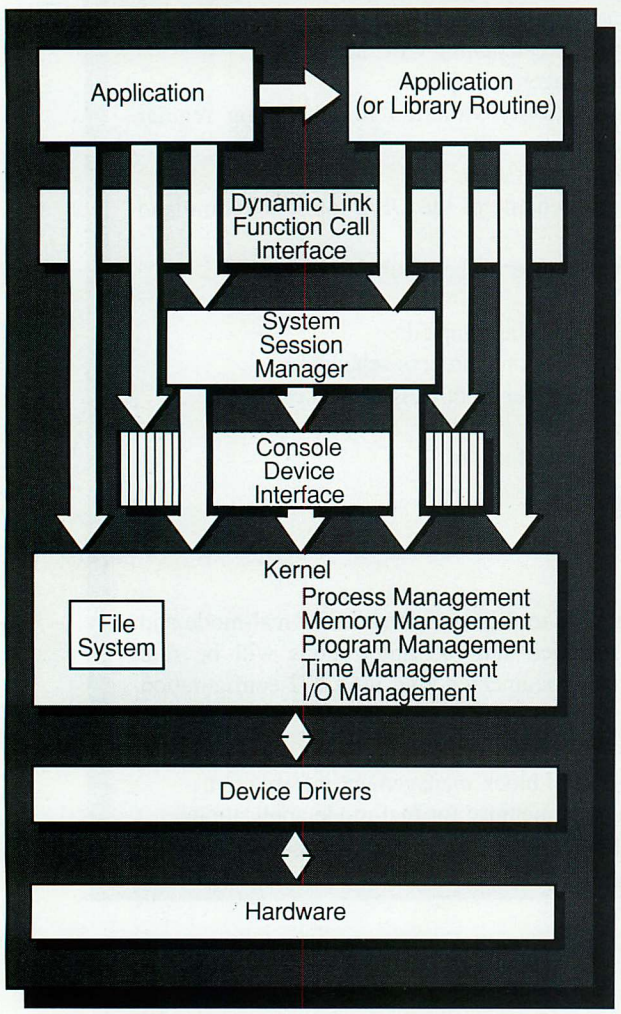
files
lastdrive

The following commands apply only to the configuration for real-mode and only if the **MODE** command specifies that old applications will be run. These real-mode-only commands are documented after the OS/2 configuration commands.

break	Check for Ctrl-Break.
fcbs	Determine file control block management information.
rmsize	Select the amount of memory for real-mode applications.
shell	Load and start the top-level command processor.



The Parallel Growth of Hardware and Software. As hardware continues to provide enhanced and more sophisticated services, the MS-DOS world will slowly give way to OS/2 and new multi-applications systems.



Block Diagram of the MS OS/2 Environment. MS OS/2 is a complicated set of interrelated modules working together to provide a complete multitasking, virtual memory environment.

around the operating system to programming with it. Instead of hunting for undocumented interrupts that perform unsupported services and bypassing the operating system to get to the hardware, developers will learn to integrate all of the tools now at their disposal. With new hardware that virtually eliminates performance bottlenecks and a new operating system that takes full advantage of that hardware, developers can focus directly on their applications.

Conclusion

Although they are not exactly right around the corner, it won't be long before we will have applications "systems" that interact with each other, work together across networks, and share resources efficiently in environments spanning micros, minis, and mainframes.

What will become of the "under-the-hood" programmer? What about all of those who still long to hack, to get their fingers dirty by directly driving hardware, manipulating code and data segments, and fooling with undocumented system calls? Well, MS OS/2 is undoubtedly full of mysterious code to decipher, but the real challenge will be fully utilizing the many tools and facilities that are now available to create state-of-the-art applications. For the serious microcomputer software developer the future is MS OS/2, and the future is now. ❑

The OS/2 Windows Presentation Manager: Microsoft Windows On the Future

Manny Vellon

Last April, Microsoft announced a series of new products, including a new operating system, Microsoft Operating System/2 (OS/2). An important part of this announcement was the inclusion of *Microsoft Windows* as a standard part of the OS/2 operating system. Currently, *Microsoft Windows* must be purchased separately and installed under MS-DOS.

The Windows presentation manager will be tightly integrated with OS/2 and will provide the same benefits that *Microsoft Windows* provides to MS-DOS: a windowed, graphical user interface and support for a variety of input and output devices. Through the presentation manager, OS/2 will replace the well-known A> prompt of MS-DOS with window-based screens.

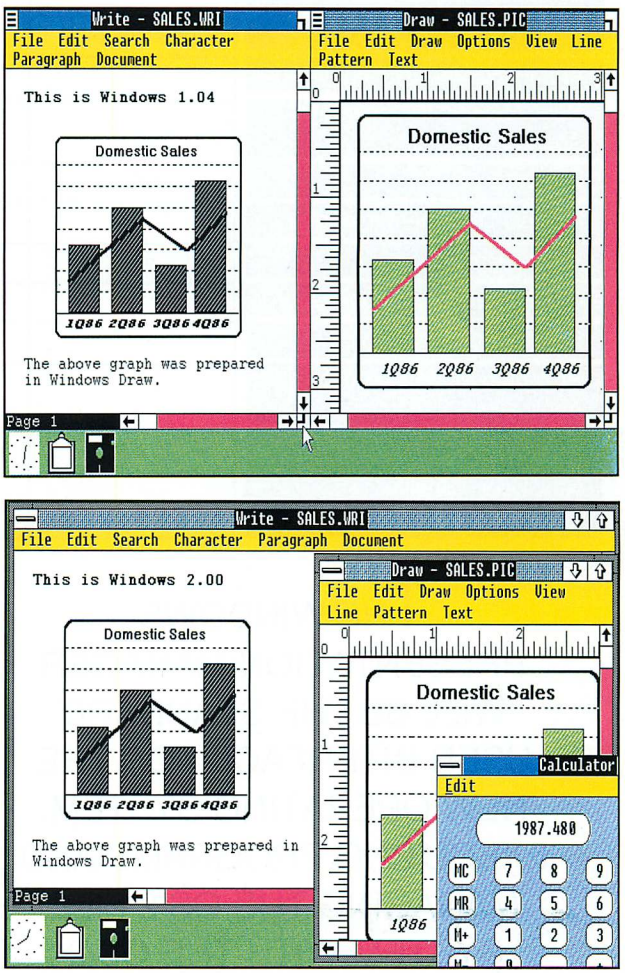
This union of *Windows* with OS/2 strengthens the role of *Windows* in future system and application products and also addresses the ease-of-use issues that have been associated with MS-DOS and IBM PC software. *Windows* provides a more intuitive interface that allows novice users to learn products more quickly.

Also important is IBM's support of *Microsoft Windows* in its new products. This support ensures that *Windows* will become a standard part of the operating environment and encourages other hardware manufacturers to support it as well.

Protected Mode

The new IBM PS/2 series, the IBM PC AT, the Compaq 386, and other Intel 286- and 386-based computers are capable of running in real mode or protected mode. In real mode, they operate much like the Intel 8088-based IBM PC: they are limited to using 1 megabyte (Mb) of memory, they can usually run only one program at a time, and they are vulnerable to crashing when a program goes awry. In protected mode, the 286- and 386-based computers don't have those limits.

Under OS/2's protected-mode operating system, programs are no longer limited to 640K; applications can take advantage of up to 16 Mb of real memory and up to 1 gigabyte of virtual memory. OS/2 is able to perform multitasking—it can run several programs simultaneously by quickly switching among them. OS/2 is more robust than real-mode operating systems; programs



▲ **Figure 1** When *Microsoft Windows* was first developed, it used tiled windows because overlapped windows were considered too slow and unusable for low-resolution displays. The new *Windows* version will employ overlapped rather than tiled windows in response to customer comments, optimized graphics algorithms, and improved processing speeds.

Figure 2: Comparison of Current and Future Windows Versions

	Microsoft Windows 1.03 & 1.04	Microsoft Windows 2.0	MS-OS/2 Windows
Presentation Spaces	Tiled	Overlapped	Overlapped
More Consistent User and Keyboard Interface	--	Yes	Yes
Processor Environments	8088 8086 286 386	8088 8086 286 386	-- -- 286 386
Large Memory Support°	--	EMS/EEMS	16Mb
Multitasking	Non Preemptive	Non Preemptive	Fully Preemptive
Enhanced Display Performance	--	Yes	Yes
Runs Existing Windows (1.03) Application	Yes	Yes	No
Graphics API °	GDI	GDI	GPI
Multiple Document Interface	--	Yes	Yes
Device Drivers	--	Enhanced	New Model
Old Application Support	--	Improved	Improved
Integral Part of OS	--	--	Yes
Protected Mode Execution Applications Execution	--	--	Yes
New Development API	--	--	Yes
New User "Shell" and Keyboard Interface	--	--	Yes

THE WINDOWS PRESENTATION MANAGER WILL BE THE STANDARD USER INTERFACE FOR THE NEW OPERATING SYSTEM. EXECUTING PROGRAMS AND MANAGING THE OS/2 FILE SYSTEM WITH THE OS/2 WINDOWS PRESENTATION MANAGER WILL BE INTUITIVE AND FAST.

run independently of each other, and if one crashes it is less likely to affect the others.

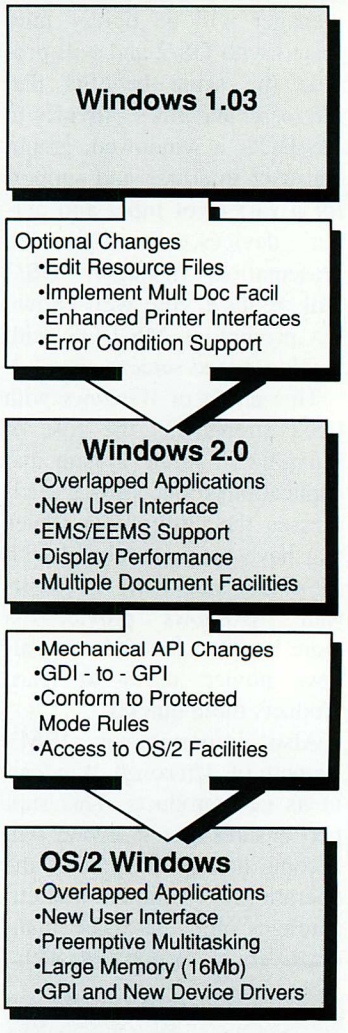
Currently, *Microsoft Windows* can also do many of these functions. By performing a variety of sophisticated functions, *Windows* can take advantage of extended memory if available and can provide rudimentary multitasking. With the OS/2 Windows presentation manager, however, these functions are much easier to implement since the underlying operating system kernel includes these capabilities.

User Interface

In addition to this symbiosis "under the hood," OS/2 Windows is integral to the operating system at the user level. The Windows presentation manager will be the standard user interface for the new operating system. Users won't have to learn about disk directories, filenames, and cryptic commands; executing programs and managing the OS/2 file system with the OS/2 Windows presentation manager will be intuitive and fast.

Although the new OS/2 Windows bears a strong resemblance to today's *Microsoft Windows,* there are some substantial differences between the two. Most significant are the

Progression for Windows-based Applications



differences in user interfaces. To eliminate these differences so as to produce a common user interface, Microsoft has announced a new version of *Microsoft Windows*—Version 2.0 (see **Figure 2**). This version, while still a real-mode version for MS-DOS, employs the same user interface to be used by the OS/2 Windows presentation manager.

When *Microsoft Windows* was first developed, it used tiled windows (see **Figure 1**). Overlapped windows were considered too slow and unusable with low-resolution displays. In order to respond to customer requests for overlapping windows, and because optimized graphics algorithms and improved processing speeds have eliminated performance bottlenecks that are found in older technology, the new *Windows* uses overlapped windows instead of tiled windows.

New *Windows* products will also have improved keyboard interfaces. Although *Windows* is best used with a mouse, it is possible, and sometimes preferable, to use it solely through the keyboard. Changes in the user interface will make this easier.

First, the keyboard interface has been improved in order to allow direct access to items in dialog boxes. In **Figure 4**, for example, the user can type Alt-F to quickly position the cursor in the filename field of the dialog box. Access to menus from the keyboard has also been enhanced by allowing the developer to select any letter from a menu command to execute the item rather than being limited to the first letter, as in the case of *Windows* Version 1.03 and Version 1.04. This helps to make it easier for software developers to provide meaningful command names while still providing fast keyboard access to commands.

```
                                        .
                                        .
                                        .
int PASCAL WinMain( hInstance, hPrevInstance, lpszCmdLine, cmdShow )
HANDLE hInstance, hPrevInstance;
LPSTR lpszCmdLine;
int cmdShow;
{
        MSG   msg;
        HWND  hWnd;
        NPWNDCLASS       pHelloClass;
                                        .
                                        .
                                        .
    /* Allocate and initialize class data structure pHelloClass */
                                        .
                                        .
                                        .
         /* Create a window class */
        if (!RegisterClass( (LPWNDCLASS)pHelloClass ) )
                return FALSE;

        /* Create an application window */
        hWnd = CreateWindow((LPSTR)"Class", (LPSTR) "Title",
                            WS_TILEDWINDOW, 0, 0, 0, 0, (HWND)NULL,
                            (HMENU)NULL, (HANDLE)hInstance,
                            (LPSTR)NULL );
                                        .
                                        .
                                        .
        /* Process messages*/
        while (GetMessage((LPMSG)&msg, NULL, 0, 0)) {
                TranslateMessage((LPMSG)&msg);
                DispatchMessage((LPMSG)&msg);
                }

                return (int)msg.wParam;
}
```

▲ **Figure 3a** **Microsoft Windows API. Programs written for MS-DOS will have to be modified to work in OS/2. Many of these changes are mechanical in nature—for example, all window manager calls are preceded with "Win". Compare the above *Windows 2.0* code with that shown in Figure 3b.**

The second significant change affects mouse operation. Currently, *Windows* employs pull-down menus. You click down on the menu bar to make visible a pop-up menu; then, while holding down the mouse button, you drag down to the desired command in the pop-up menu and let go of the button. This technique is fast, but it is prone to accidental selections and requires considerable manual dexterity. The new interface allows you to click and let go on the menu bar to make the pop-up menu visible and then click down on the desired command within the given menu.

Finally, the new interface employs some new terminology (for example, *minimize* and *maximize*, instead of *icon*

USERS WHO ARE ACCUSTOMED TO THE OLD KEYBOARD INTERFACE WON'T BE FORCED TO LEARN THE NEW ONE. *MICROSOFT WINDOWS*, VERSION 2.0 WILL WORK IN BOTH THE OLD AND NEW STYLES TO HELP USERS GET ACCUSTOMED TO THE NEW INTERFACE.

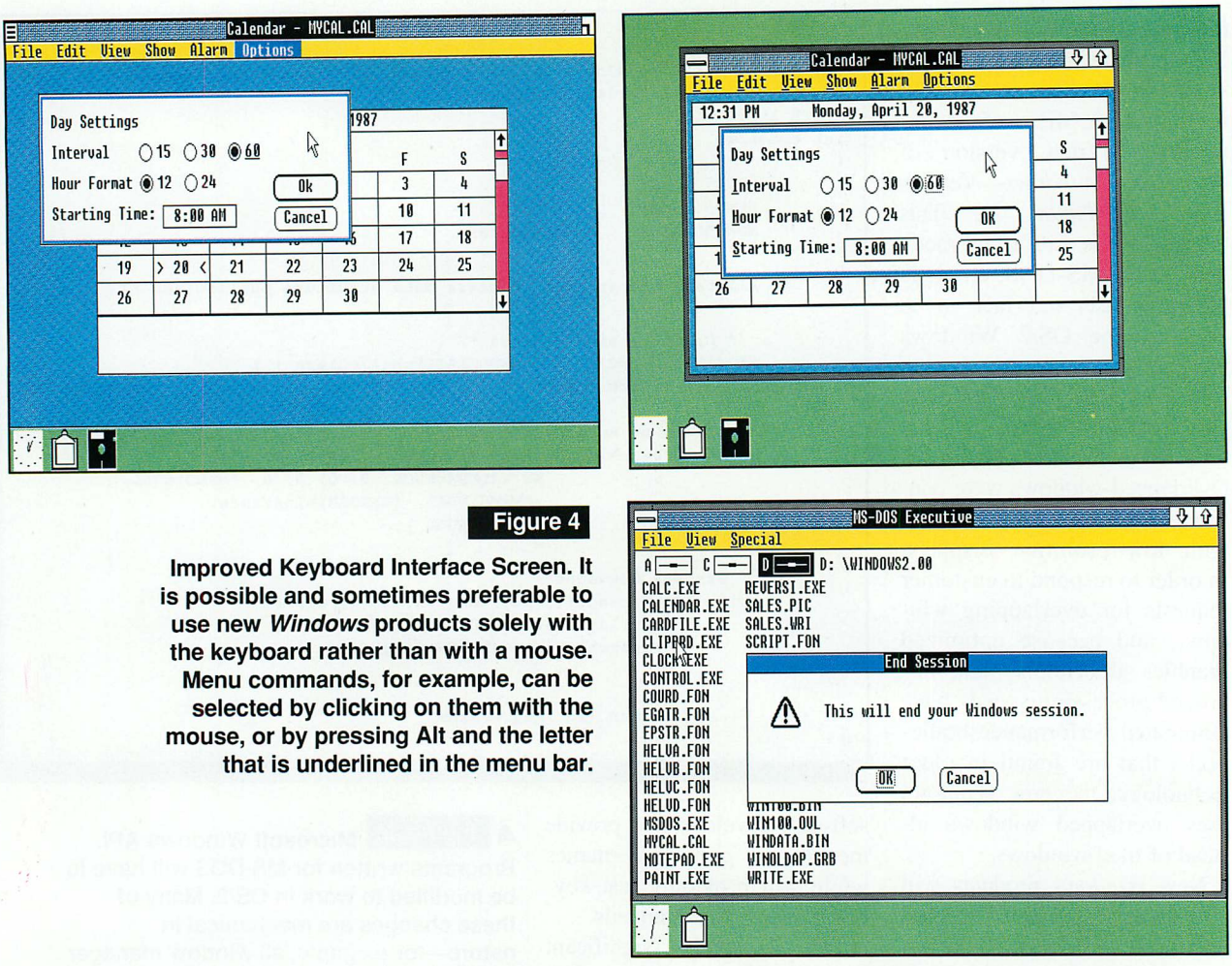


Figure 4

Improved Keyboard Interface Screen. It is possible and sometimes preferable to use new *Windows* products solely with the keyboard rather than with a mouse. Menu commands, for example, can be selected by clicking on them with the mouse, or by pressing Alt and the letter that is underlined in the menu bar.

and *zoom*) and some additional keyboard operations.

Users who are accustomed to the old keyboard interface won't be forced to learn the new one. *Microsoft Windows,* Version 2.0, will work in both the old and new styles to help users get accustomed to the new interface. However, the OS/2 Windows presentation manager will use this new interface only.

API

Although *Microsoft Windows,* Version 2.0, and the OS/2 Windows presentation manager appear to be similar and have many common functions, programs written for *Microsoft Windows* will have to be modified to work with the OS/2 Windows presentation manager. OS/2 restricts several operations in programs; for example, software interrupts are not allowed. Programs written for MS-DOS will also have to be modified to work in OS/2.

Besides the changes required by the new operating system, such as replacing software interrupts with operating system calls, other changes have been made to the OS/2 Windows Application Program Interface (API) in order to standardize coding practices, improve error handling, and exploit new graphics capabilities. Fortunately, many of these changes are "mechanical" in nature. To

follow the new coding practices, for example, all window manager calls are preceded with "Win": "WinCreateWindow", "WinShowWindow", etc. Other changes entail reordering parameters. Applications that make heavy use of graphics will require the most changes; the OS/2 Windows presentation manager contains a new graphics library that requires a variety of changes to be made to the API.

The Future

OS/2 is not intended to replace MS-DOS. Microsoft expects that real-mode (MS-DOS) and protected-mode (OS/2) products will continue to coexist until Intel 286- and 386-based computers predominate. MS-DOS and *Microsoft Windows* will continue to be developed and marketed for low-end machines, while OS/2 addresses newer, more-powerful computers. The *Windows* user interface will serve as a bridge for users, allowing them to operate each of these classes of machines regardless of the underlying hardware.

In time, Microsoft expects Intel 386-based machines to become the standard hardware environment for powerful new business and engineering applications. Microsoft will extend OS/2 to exploit the additional capabilities of the Intel 386 and will upgrade OS/2 in the future. Once more, the Windows presentation manager interface will help to keep these changes invisible to the user. ❑

```
                                        ⋮
int cdecl main(argc, argv)
int argc;
char *argv[];
{
        QMSG qmsg;
        HAB    hab;
        HMQ    hmq;
        HWND   hwnd, hwndFrame;

        /* get an anchor block */
        hab = WinInitialize();

        /* create a message queue for the application */
        hmq = WinCreateMsgQueue(hab, 0);

        /* create a window class */
        if (!WinRegisterClass(  (LPCH) "Class",
                                WndProc,
                                WS_SYNCPAINT | WS_CLIPSIBLINGS |
                                WS_SIZEREDRAW,
                                0,
                                NULL)) return FALSE;

        /* create an application window */
        hwndFrame =
                WinCreateStdWindow(NULL,
                                   FS_MENU | FS_TITLEBAR | FS_MINMAX |
                                   FS_SIZEBORDER | FS_SYSMENU,
                                   (LPSTR) "Class",
                                   (LPSTR) "Title", 0L, NULL,
                                   IDM_APPMENU,
                                   (HWND far *)&hwnd);
                                    ⋮
        /* process messages */
        while (WinGetMsg(hab, (LPQMSG)&qmsg, NULL, 0, 0))
                WinDispatchMsg(hab, (LPQMSG)&qmsg);

        /* destroy the resources used by the application */
        WinDestroyWindow(hwndFrame);
        WinDestroyMsgQueue(hmq);
        WinFinalize(hab);
}
```

▲ **Figure 3b** **OS/2 Windows presentation manager API (see Figure 3a, page 15).**

MS-DOS AND *MICROSOFT WINDOWS* WILL CONTINUE TO BE DEVELOPED AND MARKETED FOR LOW-END MACHINES, WHILE OS/2 ADDRESSES NEWER, MORE-POWERFUL COMPUTERS. THE WINDOWS USER INTERFACE WILL SERVE AS A BRIDGE FOR USERS, ALLOWING THEM TO OPERATE EACH OF THESE CLASSES OF MACHINES REGARDLESS OF THE UNDERLYING HARDWARE.

Microsoft University

In response to an often expressed desire by many programmers for a structured approach to learning the fundamentals and more-advanced features of products like *Windows,* Microsoft has expanded its educational program, which is known as Microsoft University.

Introductory and advanced classes now meet on alternate weeks in Redmond, Washington. A typical three-day session consists of lectures from 8:30 A.M. until 4:30 P.M. every day, or until 3:00 P.M. the final day. Seminar size is small, with a maximum of about 25 students, according to Linda McCarty, marketing coordinator in Microsoft's product support services group.

Brochures will be available at the end of May, and any interested people should contact the Registrar at Microsoft University. The phone number is (206) 882 - 8080. Included in these brochures are complete course descriptions and registration information.

Microsoft Operating System/2 Architecture Principles, Tools, and Practice
$1250 per person
July 20–24
August 10–14
September 14–18

MS OS/2 Device Drivers
$1000 per person
July 27–30
August 24–27
September 8–11

Microsoft Windows Application Interface for MS-DOS
$600 per person
June 17–19
July 22–24
August 5–7
August 19–21
September 2–4
September 16–18
September 30–October 2

Advanced *Microsoft Windows* Application Interface for MS-DOS
$800 per person
June 22–24
July 13–16
July 27–30
August 10–13
August 24–27
September 8–11

Debugging *Microsoft Windows* Applications
$250 per person
July 17
July 31
August 14
August 28

Overview of C for *Microsoft Windows* Development
$300 per person
July 20–21
August 3-4
August 17–18
August 31–September 1
September 14–15
September 28–29

Programming in C
$750 per person
July 13–17
August 10–14
August 31–September 4

Microsoft Macro Assembler (MASM) Assembly Language Programming
$600 per person
July 22–24
August 24–26
September 28–30

Compatibility and Transition for MS-DOS Programs

Joel Gillman

Microsoft's multitasking, virtual memory operating system has finally been unveiled. Now that MS OS/2 is a reality, programmers may be wondering with some trepidation whether they'll have to rewrite all their MS-DOS programs.

The good news is that nearly all of your existing code, both commercial products and custom-made utilities, will run just fine in OS/2's MS-DOS compatibility environment, an operating mode that emulates MS-DOS 3.3. Programs that need low-level access will not run in the compatibility mode, but these are exceptions. Most spreadsheets, word processors, and many other applications programs will work just fine.

However, once you start writing code to run in the multitasking protected mode of OS/2, you will have to do some things differently. There's a silver lining in this, because you will end up with better code once you've unlearned some of the bad habits you picked up writing in MS-DOS. Multitasking implies resource sharing, so you can't be overly greedy with system resources anymore. But what you may lose in resource access you gain in the power of multitasking.

After booting OS/2, the user is greeted by the session manager, from which the user can start any number of protected-mode applications or enter compatibility mode. For each protected-mode program, the session manager creates a new protected mode execution environment, called a *screen group*. Each screen group has a command processor (CMD.EXE, which corresponds to the MS-DOS 3.*x* COMMAND.COM), a virtual screen buffer, a virtual keyboard, and a virtual mouse, along with a virtual memory space of up to 16 megabytes (Mb) depending on the total number of screen groups.

Compatibility Mode

The MS-DOS compatibility mode in OS/2 uses the 80286 chip's real mode, which is engaged by a technique called mode switching (see "OS/2: Turning Off the Car to Change Gears," page 61). Mode switching emulates a full system reset without disrupting operation, allowing the processor to switch from protected to real mode. The compatibility mode gives the system

	DOS 3.3	OS/2 Compatibility Environment	OS/2 New Programs
Supported Hardware	8088 8086 286 386	-- -- 286 386	-- -- 286 386
Available Memory	640K	640K	16MB
Can overcommit Memory	--	--	Yes
True Multitasking	--	--	Yes
Use Software Interrupts	Yes	Yes	No
Use Hardware Interrupts	Yes	Yes	No
Use undocumented DOS Interfaces	Yes	Some	No
Have direct access to Hardware	Yes	Yes	No
Can run in the Background	No	No	Yes
Obey 286 Segment Rules	No	No	Yes

Hardware/Operating Environment Compatibility Chart. Hardware compatibility and support is different for DOS 3.3 and OS/2. OS/2 seeks to insulate applications software from the hardware.

Code	Command	Code	Command
0	Init	9	Output with Verify
3	IOCtl Input	10	Output Status
4	Input (Read)	11	Output Flush
5	Non-Destructive Input No Wait	12	IOCtl Output (Write)
6	Input Status	13	Device Open
7	Input Flush	14	Device Close
8	Output	15	Generic IOCtl

▲ **Figure 2** **Device Driver Commands. Shown here are the device driver commands supported under OS/2.**

THE MS-DOS COMPATIBILITY MODE IN OS/2 USES THE 80286 CHIP'S REAL MODE, WHICH IS ENGAGED BY A TECHNIQUE CALLED MODE SWITCHING. MODE SWITCHING EMULATES A FULL SYSTEM RESET WITHOUT DISRUPTING OEPRATION, ALLOWING THE PROCESSOR TO SWITCH FROM PROTECTED TO REAL MODE.

an 8086 interface with 1 Mb of address space and emulates MS-DOS 3.*x* with the MS-DOS SHARE utility installed. OS/2 needs up to 500 Kb of system memory in a typical configuration—substantially more than the 50 Kb required by MS-DOS 3.*x*.

The compatibility mode recognizes all of the documented MS-DOS services. A number of undocumented interrupt 21H services also exist under MS-DOS 3.*x*, but since OS/2 does not recognize most of the undocumented MS-DOS services, programs that use them won't run in the compatibility mode.

The compatibility mode supports a ROM data area and accepts service interrupts, but you cannot call ROM services directly by address. You must use the interrupt 10 through interrupt 1A services instead (see **Figure 1**). Applications can call any hardware interrupt except the CMOS clock/calendar interrupt or an interrupt already taken over by any OS/2 device driver other than the keyboard. The compatibility mode will issue interrupt 28h (spooler interrupt)—so you can still run *SideKick®*—and interrupt 24h (critical error handler).

A program running in the MS-DOS compatibility mode freezes up if the system places it in the background by switching to the protected mode. Programs frozen in the background receive no CPU service or interrupts. A program that determines the time of day by counting clock ticks, for example, will generate inaccurate times if it goes into the background.

Version-specific applications won't run in the compatibility mode, because the MS-DOS version number of OS/2 is 10. A way around this is to modify the code so that it calls DOSGETVERSION and, if it gets back 10, determines which mode it's executing by calling DOSGETMACHINEMODE. The program can then take the appropriate action.

Handling Devices

Most of your old device drivers will not run in compatibility mode. OS/2 doesn't support any of the MS-DOS block device drivers, such as those used with disk or tape drives. The only character device drivers it supports are VDI (video display interface) and CON (console) drivers, since character device drivers will work only if they are polled rather than interrupt-driven. The system supports all of the device driver command packets shown in **Figure 2.**

When you run a driver, only programs running in the compatibility mode can use its device. The device isn't available to protected-mode applications. Device drivers cannot call user code, because they operate at a higher privilege level than the user program.

Drivers are installed the same way as in MS-DOS®, using the configuration command

```
device = driver
    filename
```

MS-DOS device drivers are loaded and initialized in compatibility mode. Initialization is in most respects the same as in MS-DOS, except that no interrupt 21h (hardware-independent) functions can be performed from the initialization code.

Compatibility mode restricts which devices programs can manipulate. Sound-generating programs that need a higher frequency time base for more precise pitch control can remap the 8253 clock/timer chip, that

	DOS	OS/2		
	Old Programs	FAPI Programs		New Programs
Start With	COMMAND.COM	COMMAND.COM	CMD.EXE	CMD.EXE
Can Run in OS/2 Compatibility Box	Yes	Yes	No	No
Can Run in Background	No	No	Yes	Yes
Permit Old-Style INT 21H Dos 3.x Interrupts	Yes	No		No
Permit Undocumented Dos Interfaces	No	No		No
Have IOPL	Yes	Via FAPI		Via OS/2
Obey 286 Segment Rules	No	Yes		Yes
Can Overcommit Memory	No	No	Yes	Yes
Addressable Memory	640K	640K	16M	16M
Permit Software Interrupts	Yes	Via FAPI	No	No
Permit Hardware Interrupts	Yes	No		No
Program Residence	Below Boundary	Below Boundary	Above Boundary	Above Boundary
Permit Multitasking	No	Yes		Yes

▲ **Figure 3** **Software Compatibility Chart. Programs may run in one or more modes depending on how they behave.**

is, assign different numbers to its system interrupts. Remapping the 8259 interrupt controller is not allowed. Such programs, which must remap the 8259 in order to trap keystrokes, will not run. Applications can still hook keystrokes after OS/2 gets them, however.

Programs that need low-level disk I/O access for copy protection purposes cannot remap the disk controller. Programs do have direct access to the disk controller via interrupt 13h (floppy disk services), interrupt 25h (absolute disk read), and interrupt 26h (absolute disk write). Note that interrupt 13h and interrupt 26h are not allowed for fixed media such as hard disks.

High-speed communications applications that must remap the DMA controller won't run because the operating system remaps the controller. Applications can remap the COM, AUX, and parallel ports, although using one of these ports in the compatibility mode makes it unavailable to protected mode programs and vice versa.

80286 Restrictions

Programmers writing applications for MS-DOS developed some programming techniques and coding shortcuts to enhance performance, even though the books and manuals tended to discourage using them. Many of these techniques won't work in OS/2 because of differences between the 8086 or 8088 chip and the 80286. You'll have to observe several restrictions if you want to run your applications in the

IN ORDER TO PERMIT YOU TO WRITE NEW PROGRAMS GUARANTEED TO RUN IN MS-DOS AND IN OS/2'S PROTECTED MODE, MICROSOFT HAS DEFINED A SET OF SYSTEM CALLS THAT ARE GUARANTEED TO SUPPORT BOTH ENVIRONMENTS. THIS SET OF SYSTEM CALLS, THE FAMILY APPLICATION PROGRAM INTERFACE , IS A SUBSET OF THE FULL OS/2 APPLICATION PROGRAM INTERFACE.

compatibility environment or in protected mode.

First, don't use wrap-around segment offsets. The 8086 and 8088 microprocessors translate an out-of-range address value into something recognizable, but the 80286 doesn't. You cannot address beyond the allocated size of a segment; the system aborts the program if an offset larger than the segment descriptor's limit value is used to reference that segment.

The 80286 doesn't allow writable code segments. One of the bits in the segment descriptor identifies the segment as either code or data. A code segment's descriptor doesn't have a read/write bit, so only valid code segments can be placed in the CS (code segment selector) register, and a program may not write into valid code segments. However, an alias can be used to make a data segment into a code segment to be executed.

Since different machines use different timing speeds, don't count on the CPU clock as a timing constant. This is a typical problem in copy-protected programs.

Don't allow a division-trap handler to resume execution in the original code stream unless it is able to detect and understand differences between the 8086 or 8088 and the 80286. After a division-error trap, the 80286 points to the division instruction (including prefixes) and doesn't change the register values. The 8086 and 8088 point to the instruction following the division instruction and may change the DX:AX or AH:AL register sets as well.

The 80286 CL (low-order loop/shift/repeat count) registers won't permit shift counts greater than 31. On the 80286, the shift and rotate counts are masked to 5 bits.

When executing the PUSH SP (push stack pointer onto stack) instruction, the 80286 pushes the SP value onto the stack before incrementing the value, while the 8086 and 8088 push the SP value after incrementing it. Few programs use this particular code sequence, but for those that do, Microsoft

Some Programming Don'ts for OS/2

When writing programs for either the compatibility mode or protected mode in OS/2, you must observe a number of restraints that weren't necessary in writing for MS-DOS. Many of these restrictions arise from differences between the 8086 or 8088 and the 80286 microprocessors.

- Don't depend on segment overlap or lack of it.
- Don't depend on any relationship between segment:offset combinations and physical memory.
- Don't use wrap-around address offsets.
- Don't use the segment registers for anything but valid segment numbers.
- Never address beyond the allocated size of a segment.
- If you have to play with the I/O ports, use only the appropriate dynamic link routines (FAPI).
- Don't mix code and data in the same segment.
- Don't use undefined opcodes.
- Don't use the PUSH SP instruction.
- Don't use the POPF instruction.
- Don't use shift counts greater than 31.
- Don't use IDIV operands to produce a most-negative number.
- Don't resume execution in the original code stream after a division trap.
- Don't use redundant prefix bytes.
- Don't use CLI instructions (in protected mode).
- Don't use CPU speed as a timing constant.
- Don't examine or set explicit flag register values.
- Don't single-step interrupt instructions in debuggers.
- Don't write self-modifying code.

offers this way around the problem:

```
MOV AX,SP
PUSH AX
```

these two lines of code can be implemented by a macro.

The PUSHF instruction followed by a POPF may change the contents of the flag register in the 80286, since more flag bits are defined in the 80286 flag word than in those of the 8086 or 8088. Also, because of a bug in the 80286, the POPF may change the contents of the flag register in the 80286. You should use flag-specific instructions to set or test flag register values instead of setting or testing for explicit bit patterns.

FAPI

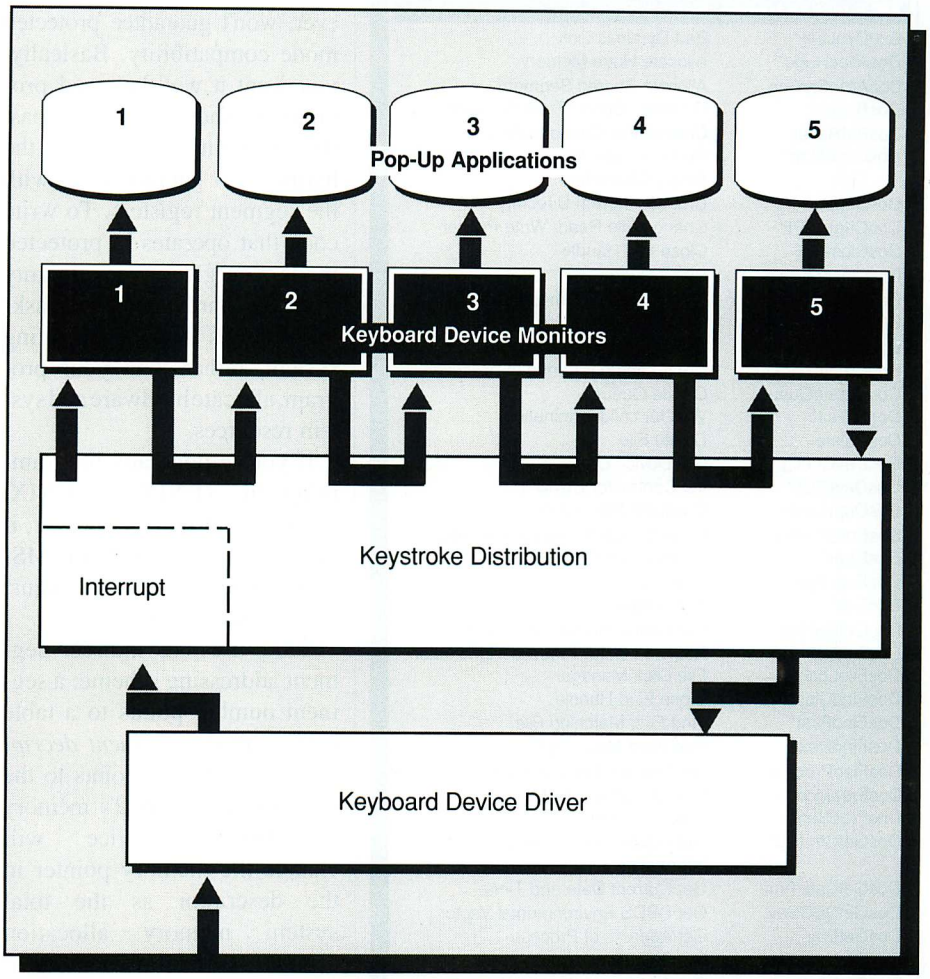
In order to permit you to write new programs guaranteed to run in MS-DOS and in OS/2's protected mode (both current and future versions), Microsoft has defined a set of system calls that are guaranteed to support both environments. This set of system calls, the Family Application Program Interface (FAPI), is a subset of the full OS/2 Application Program Interface (API).

Five types of code will run on OS/2: old programs designed for MS-DOS 3.*x* that run on OS/2 in the compatibility mode; presentation manager programs, FAPI programs that run on OS/2 in the compatibility mode; FAPI programs that run on OS/2 in protected mode; and new programs that run on OS/2 only in the protected mode. **Figure 3** summarizes the characteristics valid for each of these program types.

FAPI allows a program to be linked to run in both modes and includes system calls in all categories except those specific to protected mode, such as multitasking, run-time linking, and device monitors. Some restrictions apply to using the FAPI calls in the compatibility mode, which are discussed in detail in the *OS/2 Programmer's Guide*.

The protected mode does not give you direct access to the screen, as you would have with a MS-DOS 3.*x* BIOS call because of its memory protection. Instead, you have access to a virtual screen buffer within each screen group. The screen buffer paints the screen only when that screen group is active. For bimodal compatibility, FAPI provides a subset of the video input/output (VIO) calls for creating and writing to a virtual screen buffer.

Just using FAPI calls, how-



▲ **Figure 5 When TSR programs are run in protected mode, OS/2 assigns each a keyboard device monitor, which reads keystrokes from the keyboard. This keeps the TSRs from fighting each other to be the first to trap keystrokes.**

OS/2 API Calls

Family API (FAPI) functions are listed in blue

* Indicates that FAPI support is limited (certain restrictions are imposed)

API Function Name	Description
BadDynLink	Bad Dynamic Link
DosAllocHuge*	Allocate Huge Memory
DosAllocShrSeg	Allocate Shared Segment
DosBeep	Generate Sound From Speaker
DosBufReset	Commit File Cache Buffers
DosCaseMap*	Perform Case Mapping on String of Binary Characters
DosChdir	Change Current Directory
DosChgFilePtr	Change File Read/ Write Pointer
DosClose	Close File Handle
DosCloseQueue	Close Queue
DosCloseSem	Close System Semaphore
DosCreateCSAlias	Create CS Alias
DosCreateSem	Create System Semaphore
DosCreateThread	Create another thread of execution
DosCreateQueue	Create Queue
DosCWait*	Wait for child termination
DosDelete	Delete File
DosDevConfig	Get Device Configuration
DosDevIOCtl*	I/O Control for Devices
DosDupHandle	Duplicate File Handle
DosEnterCritSec	Enter Critical Section of Execution
DosError*	Enable Hard Error Processing
DosExecPgm*	Execute Program
DosExit*	Exit Program
DosExitCritSec	Exit Critical Section of Execution
DosExitList	Routine List for Process Termination
DosFileLocks*	File Lock Manager
DosFindClose*	Close Find Handle
DosFindFirst*	Find First Matching File
DosFindNext*	Find Next Matching File
DosFlagProcess	Set Process External Event Flag
DosFreeModule	Free Dynamic-Link Module
DosFreeSeg	Free Segment
DosGetCtryInfo*	Get Country-Dependent Formatting Information
DosGetDateTime	Get Current Date and Time
DosGetDBCSev	Get DBCS Environmental Vector
DosGetEnv	Get Address of Process
DosGetHugeShift	Get Shift Count
DosGetInfoSeg	Get Address of System Variables Segment
DosGetMachineMode	Return Current Mode of Processor
DosGetMessage	Get System Message and Insert Text Strings
DosGetModHandle	Get Dynamic-Link Module Handle
DosGetModName	Get Dynamic-Link Module Name
DosGetProcAddr	Get Dynamic-Link Procedure Address
DosGetPrty	Get Process Priority
DosGetShrSeg	Access Shared Segment
DosGetVersion	Get Dos Version Number
DosGiveSeg	Give Access to Segment
DosHoldSignal*	Disable/Enable Signal
DosInsMessage	Insert Variable Text Strings In Message
DosIOAccess	Request I/O Access to Device
DosKillProcess	Terminate Process
DosLoadModule	Load Dynamic-Link Module
DosMakePipe	Create Pipe
DosMkdir	Make Subdirectory
DosMonClose	Close Connection to OS/2 Device Driver
DosMonOpen	Open Connection to OS/2 Device Driver
DosMonRead	Read Input from Monitor Structure
DosMonReg	Register Set of Buffers as Monitor
DosMonWrite	Write Output to Monitor Structure
DosMove	Move File or Subdirectory

CONTINUED

ever, won't guarantee protected mode compatibility. Basically, you want a well-behaved program; it shouldn't sneak past the operating system to the hardware or get too clever with the segment registers. To write code that operates in protected mode, you give up sovereignty over the hardware. Multitasking requires that the operating system, rather than your program, allocate hardware and system resources.

If you've done any programming in XENIX or UNIX, you're used to this. However, if you've only worked in MS-DOS, you'll want to adjust your thinking a little.

OS/2 uses an indirect segment addressing scheme: a segment number points to a table entry, called a *segment decriptor,* which in turn points to the memory space. OS/2's memory management service will change the memory pointer in the descriptor as the total system memory allocation changes, so there is no way of knowing where a given segment number actually puts you in physical memory. Thus, a well-behaved program doesn't try to interpret segment numbers or calculate other segment locations from a given segment number.

You can't assume that any given segments overlap or don't overlap, nor can you assume any particular relationship between segment:offset combinations and physical memory. The segment number is nothing more than a segment ID, with no particular significance apart from that.

The segment registers are intended for the storage of valid segment numbers. If you store invalid numbers there—for example, by using the segment registers as scratch-pad memory (the 8086 doesn't seem to have enough registers to suit some people)—your program will crash.

Your program cannot issue a CLI (clear interrupt) instruction in protected mode because this causes a protection trap. When in compatibility mode, IRET (interrupt return) restores the previous value of IFLG (interrupt flag), but IRET has no effect in protected mode. Protected mode programs can interact directly with hardware only by linking to a special I/O Privilege Level Segment. This allows access to the 80286 processor's Ring 2 security level.

Bimodal Device Drivers

OS/2 supports bimodal device drivers that run in either mode, obviating the need for the system to switch modes to process interrupts.

Unlike a MS-DOS 3.*x* device driver, a bimodal OS/2 driver has to support multiple synchronous and asynchronous requests. However, the basic structure remains pretty much the same: the driver contains a strategy routine and an interrupt routine. In addition, some device drivers may have to include a routine to trap ROM BIOS software interrupts from compatibility mode.

An application program uses a request packet to call the strategy routine, just as with a MS-DOS 3.*x* driver. The strategy routine determines whether a request is valid and places valid requests in a queue to the device, using the DevHlp functions to manage the queue. If the device is idle, the strategy routine starts it and returns to the operating system, which suspends the thread until the

request has been executed.

When the device completion interrupt occurs, the interrupt routine sets the return status in the request packet and removes that packet from the queue. It then calls a DevHlp routine, DevDone, to tell OS/2 that the request is complete.

The strategy routine should disable interrupts when checking to see if the device is active and when examining the queue. This protects the interrupt routine from other driver request interrupts. When interrupts are reenabled, the interrupt routine will receive only ones of higher priority.

This is only a sample of some design considerations in writing OS/2 bimodal device drivers. The subject is treated at greater length in the *OS/2 Device Driver Guide*.

New Tools

A new C compiler that runs in both modes is included in the *OS/2 Software Development Kit*. However, executable files written in *Microsoft C Compiler*, Version 4.0, should have no trouble running in the compatibility mode, as long as they meet the general criteria outlined earlier for operation in OS/2. You don't need to recompile or relink them with the new OS/2 C run-time library.

The new C library is nearly identical to the Version 4.0 library, which is designed for single-thread execution only. Most of the functions are not re-entrant and therefore will not work in a multithread process. **Figure 4** lists the re-entrant functions that may be used in multithread programs.

All the routines in **Figure 4** will work properly in programs that use a far data model (compact and large model). However, in near-data-model-memory (small- and medium-model) programs, only some of these routines are guaranteed to function properly. The others have model-dependent pointers in their interfaces and have the potential to allocate stack space outside the default segment (SS!=DS).

The kit also contains a new macro assembler that will run in protected mode. Again, existing MASM executable files will run in the compatibility mode, subject to the general restrictions described earlier.

The MS-DOS 3.2 network function calls aren't supported in the compatibility mode, but the new Microsoft OS/2 LAN Manager will allow networking in protected mode.

Device Monitors

One problem that has plagued MS-DOS programmers is that when several terminate - and - stay - resident (TSR) programs are loaded into memory, they tend to step on each other in order to catch a reactivate keystroke from the keyboard. When you put a TSR like Borland's *SideKick*, Rosesoft's *ProKey*, or North Edge Software's *TimeSlips* into the background, the program calls interrupt 21h function 31h (Terminate and Stay Resident) and remains in memory even though you return to the system prompt. The program is still watching the keyboard, trapping keystrokes before the system gets them. When you hit a hot key combination, such as Ctrl-Alt, the program pops back onto the screen. The problem is that when two or more pop-up programs reside in the background at the same time, each wants to trap the

CONTINUED

Family API (FAPI) functions are listed in blue

* Indicates that FAPI support is limited (certain restrictions are imposed)

API Function Name	Description
DosMuxSemWait	Wait for one of n semaphores to be cleared
DosNewSize	Change File Size
DosOpen*	Open File
DosOpenQueue	Open Queue
DosOpenSem	Open Exisiting Semaphore
DosPeekQueue	Peek Queue
DosPurgeQueue	Purge Queue
DosPutMessage	Output Message Text To Handle
DosQCurDir	Query Current Directory
DosQCurDisk	Query Current Disk
DosQFHandState	Query File Handle State
DosQFileInfo*	Query File Information
DosQFileMode	Query File Mode
DosQFSInfo	Query File System Information
DosQHandType	Query Handle Type
DosQueryQueue	Query Queue
DosQVerify	Query Verify Setting
DosRead	Read from File
DosReadAsync	Asynchronous Read from File
DosReadQueue	Read from Queue
DosReAllocHuge*	Change Huge Memory Size
DosReAllocateSeg*	Change segment Size
DosResumeThread	Restart Thread
DosRmdir	Remove Subdirectory
DosSelectDisk	Select Default Drive
DosSemClear	Clear (release) Semaphore
DosSemRequest	Request Semaphore
DosSemSet	Set Semaphore Owned
DosSemSetWait	Set Semaphore and wait for next Clear
DosSemWait	Wait for Semaphore to be Cleared
DosSetDateTime	Set Current Date and Time
DosSetFHandState*	Set File Handle State
DosSetFileInfo	Set File Information
DosSetFileMode	Set File Mode
DosSetMaxFH	Set Maximum File Handles
DosSetPrty	Set Process Priority
DosSetSigHandler*	Handle Signal
DosSetVec	Establish Handler For Exception Vector
DosSetVerify	Set/Reset Verify Switch
	Delay Process Execution
DosSubAlloc	Suballocate Memory within Segment
DosSubFree	Free Memory Suballocated within Space
DosSubSet	Initialize or Set Allocated Memory
DosSuspendThread	Suspend Thread Execution
DosSystemService	Dos System Process Services
DosTimerAsync	Start Asynchronous Timer
DosTimerStart	Start Periodic Interval Timer
DosTimerStop	Stop Asynchronous or Interval Timer
DosPTrace	Interface for Program Debugging
DosWrite	Synchronous Write to File
DosWriteAsync	Asynchronous Write to File
DosWriteQueue	Write to Queue
KbdCharIn	Read Character Scan Code
KbdFlushBuffer	Flush Keyboard Buffer
KbdGetStatus	Get Keyboard Status
KbdPeek*	Peek at Character-Scan Code
KbdRegister	Register keyboard Subsystem
KbdSetStatus	Set Keyboard Status
KbdStringIn	Read Character String
MouClose	Close Mouse Device For Current Screen Group
MouDrawPtr	Release Screen Area For Device Driver Use

CONTINUED

OS/2 API Calls

CONTINUED

Family API (FAPI) functions are listed in blue

* Indicates that FAPI support is limited (certain restrictions are imposed)

API Function Name	Description
MouGetDevStatus	Get Current Pointing Device Driver Status Flags
MouGetEventMask	Get Current Pointing Device One-Word Event Mask
MouGetNumButtons	Get Number of Buttons
MouGetNumMickeys	Get Number of Mickeys-Per-Centimeter
MouGetNumQueel	Get Current Status for Pointing Device Event Queue
MouGetScaleFact	Sets Scale Factors for Current Pointing Device
MouOpen	Open Mouse Device For Current Screen Group
MouReadEventQue	Read Pointing Device Event Queue
MouRegister	Register Mouse Subsystem
MouRemovePtr	Reserve Screen Area For Application Use
MouSetEventMask	Assign New Event Mask To Current Pointing Device
MouSetHotKey	Set System Hot Key
MouSetPtrShape	Set Pointer Shape and Size
MouSetScaleFact	Set Scale Factors for Current Positioning Device
VioEndPopUp	Deallocate a Pop-up Display Screen
VioGetAnsi	Get ANSI State
VioGetBuf	Get Logical Video Buffer
VioGetCurPos	Get Cursor Position
VioGetCurType	Get Cursor Type
VioGetPhysBuf	Get Physical Video Buffer
VioPopUp	Allocate Pop-up DisplayScreen
VioPrtScreen	Print Screen
VioReadCellStr	Read Character-Attribute String
VioReadCharStr	Read Character String
VioRegister	Register Video Subsystem
VioSavReDrawWait	Screen Save RedrawWait
VioScrLock*	Lock Screen
VioScrollDn	Scroll Screen Down
VioScrollLf	Scroll Screen Left
VioScrollRt	Scroll Screen Right
VioScrollUp	Scroll Screen Up
VioScrUnLock	Unlock Screen
VioSetAnsi	Set ANSI On or Off
VioSetCurPos	Set Cursor Position
VioSetCurType	Set Cursor Type
VioSetMode	Set Display Mode
VioShowBuf	Display Logical Buffer
VioWrtCellStr	Write Character-Attribute String
VioWrtCharStr	Write Character String
VioWrtCharStrAttr	Write Character String With Attribute
VioWrtNAttr	Write N Attributes
VioWrtNCell	Write N Character-Attributes
VioWrtNChar	Write N Characters
VioWrtTty	Write TTY String

keyboard interrupt before any other program does (see "Moving Toward an Industry Standard for Developing TSRs," *Microsoft Systems Journal*, Vol. 1, No. 2).

OS/2 solves this problem in protected mode by giving each TSR its own keyboard device monitor for reading keystrokes. Suppose you want to have several TSRs written by different vendors available within a given screen group. Each one will have its own keyboard device monitor. The monitors receive keystrokes in the order of monitor registration, which is set when the programs are first run. Keystrokes are passed on to the first monitor registered, which can trap the keystroke and generate a response or pass the keystroke on the next monitor registered (see **Figure 5**).

Obviously, in this scheme, no two TSRs can use the same reactivate-key sequence. You can't reactivate a pop-up program in one screen group from a different screen group. By definition, a screen group is made up of a virtual screen, virtual mouse, virtual keyboard, and virtual memory space, and screen groups can't talk to each other.

The Tradeoff

The constraints that OS/2 imposes on the programmer may seem at first rather strict. With MS-DOS, you are pretty much free to use the system services or to leave them alone and go directly to the CPU instead. But in a multitasking system, you just can't do that. OS/2 takes over the CPU and the hardware, granting access only in certain instances.

On the other hand, a lot of old problems go away. Applications will no longer have to fight each other for system resources, because OS/2 allocates resources among them on a priority basis. Applications will have a common program interface (API) to work with, ensuring future compatibility. What you lose in resource access you gain in the new opportunities of multitasking, such as the potential for real-time multithread applications.

Similarly, the memory access restrictions are compensated for in protected mode by the greatly increased size of the memory space that you can work with and the addition of memory management capabilities. The only limitation is that you must use system calls for access rather than stuffing numbers directly into the segment registers. And the 80286 chip is fast enough that efficiency issues aren't as critical as they were on the original PCs and XTs, so programming shortcuts such as segment games just aren't necessary anymore. ❑

OS/2 Multitasking: Exploiting the Protected Mode of the 80286

Ray Duncan

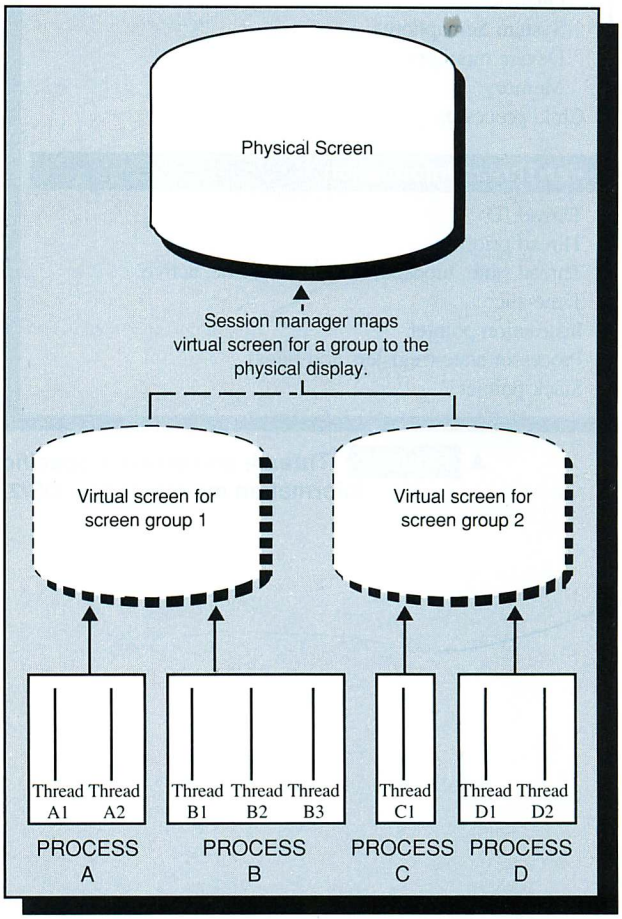
Multitasking is the technique of dividing CPU time between multiple tasks so that they appear to be running simultaneously. Of course, the microprocessor is only executing one sequence of machine instructions within one task at any given time, but the switching from one task to another is invisible to both the user and the programs themselves. The operating system is responsible for allocating system resources such as memory to the various executing tasks and for resolving contending requests to such peripherals as video displays and disk drives.

The part of the operating system that allocates CPU time between tasks is called the *scheduler* or *dispatcher*, and the rotation from one task to another, or from a task to a module of the operating system, is called a *context switch*. When a context switch occurs, the dispatcher must save the current state of the task that was executing, including its registers and program counter, load the registers and program counter belonging to the next task to run, and transfer control to that task at the point where it was previously suspended.

There are two basic strategies for task scheduling that are used by modern multitasking operating systems: *event-driven* and *pre-emptive*.

Event-driven schedulers rely on each task to be "well-behaved" and yield control at frequent enough intervals so that every program has acceptable throughput and none will be "starved" for CPU cycles. This yielding may be explicit (the program calls a specific operating system function to give up control) or implicit (the program is suspended when it requests the operating system to perform I/O on its behalf and regains control only after the I/O is completed and other tasks have in turn yielded control). This strategy is quite efficient in transaction-oriented systems where there is a great deal of I/O and not much computation, but the system can be brought to its knees by a single compute-bound task, such as an in-memory sort.

A pre-emptive scheduler relies on the presence of an external signal generated at regular intervals, typically a hardware interrupt triggered by a real-time clock. When the interrupt occurs, the operating system



Three types of system objects are involved in OS/2 multitasking: processes, threads, and screen groups. A process represents an application program accessing system resources such as files, memory, and inter-process communication facilities. A process can contain multiple concurrent points of execution called threads; each thread has its own priority and stack. Processes are collected into screen groups that read and write a virtual display and keyboard; theSession Manager is used to select a screen group.

Ray Duncan is a regular columnist for Dr. Dobb's Journal of Software Tools *and author of the best-selling* Advanced MS-DOS, *published by Microsoft Press.*

Per-Process Information

Process ID (PID)
Disk-swapping information
Local Descriptor Table (LDT) pointer
System resources owned or opened:
  Files
  Pipes
  Queues
  System Semaphores
  Device monitors
  Memory
Child processes

Per-Thread Information

Thread ID
Thread priority
Thread state: blocked, ready to execute, active
Time-slice
Instruction pointer
Processor state (registers and flags)
Stack pointer

▲ **Figure 1** **Thread- and process-specific information maintained by OS/2.**

gains control from whatever task was executing, saves its context, evaluates the list of programs that are ready to run, and gives control to (dispatches) the next program. This approach to multitasking is often called "time-slicing"; this term is derived from the mental image of dividing the sweep of a second hand around a clock face into little wedges and doling out the pieces to all the eligible programs.

Modern mainframe and minicomputers, and the more powerful microcomputers, such as the Intel 80286 and Motorola 68020, include hardware features specifically designed to make multitasking operating systems more efficient and robust. These include privilege levels and memory protection.

In the simplest use of privilege levels, the CPU is either in kernel mode or user mode at any given time. In kernel mode, which is reserved for the operating system, any machine instruction can be executed, and any location in memory can be accessed. As part of the mechanism of transferring control from the operating system to an application program, the CPU is switched into user mode. In this CPU state, any attempt to execute certain reserved instructions, such as writing to an I/O port, causes a hardware interrupt and returns control to the operating system. (Although the Intel 80286 microprocessor actually supports four privilege levels, OS/2 ordinarily uses only the highest and lowest of these.) Similarly, the hardware memory protection mechanisms detect any attempt by an application program to access memory that does not belong to it, and generate a hardware interrupt that allows the operating system to abort the offending task.

Microsoft OS/2 is designed around a pre-emptive, priority-based multitasking scheduler. Understanding OS/2 multitasking requires a grasp of three distinct, but related, concepts or system objects: processes, threads, and screen groups.

Processes

The simplest case of an OS/2 process is very similar to an application program loaded for execution under MS-DOS 2.*x* and 3.*x*. A process is started (whether by a shell or command processor or by another application) by a call to the OS/2 service DOSEXECPGM. OS/2 initiates a new process by allocating memory segments to hold that process's code, data, and stack, and then initializing the memory segments from the contents of the program's .EXE disk file.

Once it is running, a process can obtain access to additional system resources such as files, pipes, semaphores, queues, and additional memory with various OS/2 function calls (see **Figure 1**). A process terminates itself with a call to the OS/2 function DOSEXIT and can also be aborted by its parent process, by an unrecoverable hardware error, or by a memory protection fault. In any case, OS/2 will automatically release all resources belonging to the process when it terminates.

Ordinarily, processes are only aware of themselves, OS/2 services, and any child processes they start directly. They cannot directly access memory space or resources belonging to another process,

including a child process, without the cooperation of that process.

Threads

The MS OS/2 scheduler, however, knows nothing about processes; it partitions the available CPU cycles among dispatchable entities known as *threads*. A thread has a point of execution, a priority, a stack pointer, and general register contents (see **Figure 1**). At any given time, a thread is either blocked (waiting for I/O or some other event), ready to execute, or actively executing (it has control of the CPU).

Each process has a primary thread that receives control from OS/2 when the process is started; it begins executing at the program's designated entry point. However, that primary thread can start up additional threads within the same process. Multiple threads within a process execute asynchronously to one another, can have different priorities, and can manipulate one another's priorities.

Although the threads within a process have separate stacks, they share the same near data segment (DGROUP) and thus the same local heap. Carefully designing code to use the stack for local variables allows procedures to be shared between threads—this happens naturally in C programs. Access to static variables or other data structures must be coordinated between threads through the use of RAM semaphores or similar mechanisms. The threads also share all the other system resources owned by the process as a whole—open files, system semaphores, queues, and pipes—but OS/2 generally provides automatic serialization of operations on these resources.

When a thread within an application program is executing, the system is in user mode; the thread can only access the resources owned by the process that contains it and cannot execute certain machine instructions. A clock tick, another hardware interrupt, or a

EACH PROCESS HAS A PRIMARY THREAD THAT RECEIVES CONTROL FROM OS/2 WHEN THE PROCESS IS STARTED; IT BEGINS EXECUTING AT THE PROGRAM'S DESIGNATED ENTRY POINT. HOWEVER, THAT PRIMARY THREAD CAN START UP ADDITIONAL THREADS WITHIN THE SAME PROCESS. MULTIPLE THREADS WITHIN A PROCESS EXECUTE ASYNCHRONOUSLY TO ONE ANOTHER AND CAN HAVE DIFFERENT PRIORITIES.

Configuring the OS/2 Multitasker

There are four different directives that can be placed in the system CONFIG.SYS file to influence the operation of OS/2 multitasking. These are

```
THREADS=n
MAXWAIT=seconds
PRIORITY=ABSOLUTE|
         DYNAMIC
MAXWAIT=x[,y]
```

The THREADS directive controls how many threads can be simultaneously created in the system. The parameter *n* may be 16–255, with a default of 16, which is sufficient for OS/2 and a few simple processes. The upper limit of 255 cannot be expanded.

When a thread is denied the CPU for the number of seconds specified by MAXWAIT because of other higher-priority threads using up all the time-slices, the starved thread receives a temporary increase in its priority for one time-slice.

The PRIORITY directive activates OS/2's mechanisms for dynamically altering the priority of each thread based on the activity of other threads within the system. When PRIORITY=ABSOLUTE, the MAXWAIT directive has no effect whatsoever.

The TIMESLICE directive controls the length of the time-slices used by the OS/2 scheduler. The *x* parameter represents the normal length of a thread's time-slice in milliseconds. If a thread uses up its entire time-slice and must be pre-empted, its next time-slice will be one tick longer, up to the limit given by the *y* parameter. This strategy is used by OS/2 to reduce the amount of context-switching overhead when several compute bound threads are running.

PROCESSES ARE IN TURN MEMBERS OF SCREEN GROUPS. WHEN THE USER PRESSES THE SYSREQ KEY, HE EXITS FROM THE CURRENT SCREEN GROUP TO A MENU DISPLAYED BY THE SESSION MANAGER. HE CAN THEN SELECT A PROGRAM ALREADY EXECUTING IN ANOTHER SCREEN GROUP OR ESTABLISH ANOTHER SCREEN GROUP BY LOADING A NEW COPY OF THE COMMAND PROCESSOR.

call by the application for an OS/2 function causes a transition back into kernel mode, so that OS/2 can service the interrupt or provide the requested operation.

When OS/2 is ready to exit kernel mode, the scheduler receives control and examines its list of active threads. The thread with the highest priority that is ready to execute gains control of the machine. If the thread that was just interrupted is one of several eligible threads and has not used up its time-slice, it receives preference. If a thread becomes starved for CPU cycles because other threads with higher priorities are getting all the attention, OS/2 will temporarily bump that thread's priority to a higher value.

Screen Groups

Processes are in turn members of *screen groups*, which is what the average user perceives as being OS/2 multitasking. When the user presses the SysReq key, he exits from the current session, or screen group, to a menu displayed by the Session Manager. He can then select a command processor or other program already executing in another screen group or establish a new screen group by loading a new copy of the protected-mode command processor.

OS/2 maintains a separate *virtual screen* for each screen group, which receives the output of all the processes in that group. The virtual screen is mapped to the physical display whenever that screen group is selected by the user with the Session Manager. New processes are added to a screen group by being "spawned" by the command processor or another process already executing within the group.

By convention, only one of the processes within a screen group should be in the foreground at any given time, that is, writing to the screen and reading from the keyboard. The OS/2 DETACH command allows programs to be placed in the background from the command processor level. However, since OS/2 does not place any restrictions on access to the virtual display by the various processes within a screen group, DETACHing normal programs is not usually too useful since it just results in chaotic displays. Programs intended to be used as background tasks must be specially designed to use the OS/2 services for pop-up windows and keyboard monitors, so that they do not disrupt the displays or otherwise interfere with the proper operation of the foreground task within their group.

A screen group, or session,

Process Control	
DOSEXECPGM	Load and execute a child process
DOSCWAIT	Wait for child process to terminate
DOSKILLPROCESS	Unconditionally terminate another process.
DOSPTRACE	Inspect/modify/trace a child process
Threads Controlling Threads	
DOSCREATETHREAD	Create another execution thread within the sameprocess
DOSSUSPENDTHREAD	Suspend the execution of a thread
DOSRESUMETHREAD	Reactivate a thread
DOSEXIT	Terminate current thread or all threads in process
Read/Alter Thread Priorities	
DOSGETPRTY	Get the priority of specified thread
DOSSETPRTY	Set the priority of specified thread
Inter-Thread Protection	
DOSENTERCRITSEC	Disable other threads in same process
DOSEXITCRITSEC	Re-enable other threads in same process

▲ **Figure 2** **OS/2 multitasking services at a glance.**

is removed from the system by first terminating any active application programs within that group and returning to the command processor prompt. Then, entry of the EXIT command terminates the command processor itself. The Session Manager regains control and displays a menu of the remaining screen groups.

OS/2 Programming

OS/2 offers a diverse set of services to application programs that allow the creation of complex and powerful multitasking applications (see **Figure 2**). These services include

- starting and stopping child processes
- obtaining the return code of a child process
- starting, suspending, and destroying threads
- altering the priorities of threads
- inter-process communication (see "OS/2 Inter-Process Communication: Semaphores, Pipes, and Queues," page 37)

OS/2 DOSEXECPGM is used by the *parent process* to load and execute the *child process*. This function is analogous to, but considerably more powerful than, the EXEC function (Int 21h Function 4Bh) that was available in MS-DOS 2.*x* and 3.*x*. The child process can, in turn, load other processes and so on until system limits on threads or open files are exhausted or the system runs out of swap space on the disk.

The OS/2 DOSEXECPGM function is called with

- the address of the filename of the child process to be executed
- a flag controlling whether the child process is synchronous or asynchronous

```
                    :
          push  ds                  ; address of object buffer
          push  offset DGROUP:ObjName
          push  ObjNameLen          ; length of object buffer
          push  0                   ; execute synchronously
          push  ds                  ; address of argument block
          push  offset DGROUP;ArgBlk
          push  0                   ; address of envir.block
          push  0                   ; (OL=inherit parent's)
          push  ds                  ; address to receive return
          push  offset DGROUP:RetCode  ;and termination codes
          push  ds                  ; address of name of child
          push  offset DGROUP :PgmName
          call  DOSEXECPGM          ; transfer to 286DOS
          or    ax,ax               ;was EXEC successful?
          jnz   error               ;jump if EXEC failed...
                    :
ObjName     db   64 dup (0)     ; receives name of dynamic link
ObjNameLen  equ  $-ObjName      ; causing EXEC failre

ArgBlk      db   'chkdsk *.*',0 ; block of argument strings for child...
            db   0              ; extra null byte terminates block

                                ; receives return codes from child...
RetCode     dw   0              ; termination code for child
            dw   0              ; result code from child's DOSEXIT

PgmName     db   'chkdsk.eve',0 ;name of child program to run
```

▲ **Figure 3** **This sample code fragment demonstrates the use of the OS/2 system service DOSEXECPGM to run CHKDSK as a synchronous child process.**

- the addresses of an argument string block and an environment block to be passed to the child process (Each of these blocks consists of a series of null-terminated (ASCIIZ) strings; the block is terminated by an extra null byte. The environment block corresponds exactly to the environment block you are familiar with in MS-DOS 2.*x* and 3.*x*, while the simplest case of the argument block is simply a copy of the command line that invoked a process.)
- addresses of buffers to receive the process ID of the child and other information

When a child process executes synchronously, execution of the thread in the parent process that made the DOSEXECPGM call is sus-

A SCREEN GROUP IS REMOVED FROM THE SYSTEM BY FIRST TERMINATING ANY ACTIVE APPLICATION PROGRAMS WITHIN THAT GROUP. THEN, ENTRY OF THE EXIT COMMAND TERMINATES THE COMMAND PROCESSOR ITSELF. THE SESSION MANAGER REGAINS CONTROL AND DISPLAYS A MENU OF THE REMAINING SCREEN GROUPS.

pended until the child process terminates, either intentionally or owing to an error condition. When the thread in the parent resumes execution, it is supplied with the return code of the child and a termination code that indicates whether the child terminated normally or was aborted by the operating system.

If the child process is asynchronous, all threads in the parent process continue to execute while the child is running. Any thread in the parent process can later use the DOSCWAIT call to resynchronize with the child process by suspending itself until the child terminates and then obtaining its return code. As an alternative, the parent can use the process ID of the child, which is supplied by OS/2 on return from the original DOSEXECPGM call, to unilaterally abort execution of the child process with the DOSKILLPROCESS call at any time. (A special usage of the asynchronous option of DOSEXECPGM, together with the DOSPTRACE function, allows the child program to be traced, inspected, and modified under the parent program's control. This combination of OS/2 services allows the creation of program debuggers that are compatible with 80286 memory protection.)

The child process automatically inherits access to certain resources of the parent process. These resources include the handles for any open files (unless the parent process explicitly opened the files with the noninheritance flag), handles to any open pipes, handles to any open system semaphores (but not ownership of the semaphores), and a copy of the parent's environment block (unless the parent goes to the trouble of creating a new block and passes a pointer to it).

Figure 3 is a demonstration of the DOSEXECPGM function—a synchronous execution of CHKDSK as a child process. The various options for asynchronous execution, coupled with several options that are available with the DOSCWAIT function and the DOSKILLPROCESS function, allow for very flexible execution-time relationships between parent and child processes.

Managing Threads

OS/2 has a rich repertoire of function calls for control of multiple threads of execution within a process. The use of multiple threads is particularly appropriate for cases in which a process must manage several I/O devices with vastly different I/O rates (for example, the keyboard, disk, and video display) and remain rapidly responsive to the needs of each. Multiple threads are also appropriate for cases in which a process needs to run in multiple instances, but the instances do not require separate data segments or resources, since multiple threads are started much more quickly than multiple processes and have less system overhead.

As mentioned previously, each process has a single primary thread that is known to the OS/2 dispatcher when it is started up, and this thread's initial point of execution is the program's designated entry point. The primary thread can use the MS OS/2 function DOSCREATETHREAD to start up additional threads within the process, and those new threads can also start up threads and so on. Each thread is initially entered through a far call from OS/2 and can terminate through a far return or by issuing the OS/2 function call DOSEXIT. (A process is terminated when the sole remaining active thread in a process issues DOSEXIT, or when any thread issues DOSEXIT with a special parameter that indicates that all threads should be immediately terminated.)

A thread can use the function DOSSLEEP to suspend itself for a programmed period of time, or it can block on a semaphore to await reactivation by another thread or process triggering that same semaphore. Alternatively, a thread can use the functions DOSSUSPENDTHREAD or DOSRESUMETHREAD to suspend or reactivate other threads within the same process without their cooperation. Similarly, a thread can use the functions DOSGETPRTY or DOSSETPRTY to inspect or modify the execution priority of itself or other threads in accordance with execution-time requirements.

Because a thread can be unilaterally suspended by another thread without its knowledge or cooperation, the functions DOSENTERCRITSEC and DOSEXITCRITSEC are provided in order to protect a thread from interruption while it is executing a critical section of its code.

Figure 4 demonstrates the use of DOSCREATETHREAD by one thread to start up another thread that emits ten beeps at one-second intervals and then terminates. Although this is a trivial use of multiple threads, it gives an inkling of the enormous power of this concept and the ease with which asynchronous processing

can be incorporated into an OS/2 application.

Summary

OS/2 uses a time-slicing, preemptive, priority-based multitasking strategy. The Intel 80286 microprocessor's support for privilege levels and memory protection are fully exploited by OS/2 in order to run multiple concurrent tasks and to protect those tasks from damaging each other or the operating system.

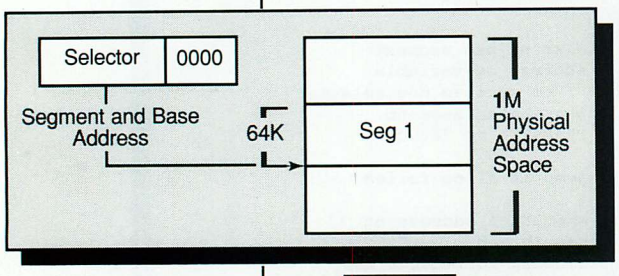
A process represents the execution of an application and the ownership of any resources—files, memory, etc.—associated with that execution. Processes can spawn other processes and can exert certain control over those child processes, but sharing of data between two processes is possible only with the cooperation of both processes. OS/2 has a wealth of facilities for inter-process communication, which are discussed in "OS/2 Inter-Process Communication: Semaphores, Pipes, and Queues."

The thread is the dispatchable element used by OS/2 to track execution cycles of the processor. Threads can start, stop, and influence the execution of other threads within the same process. Sharing of data between threads is natural and, in fact, difficult to avoid, since all threads in a process share access to the same memory, files, pipes, queues, and semaphores. In essence, the OS/2 loader knows about processes, but the OS/2 scheduler knows about threads. ❑

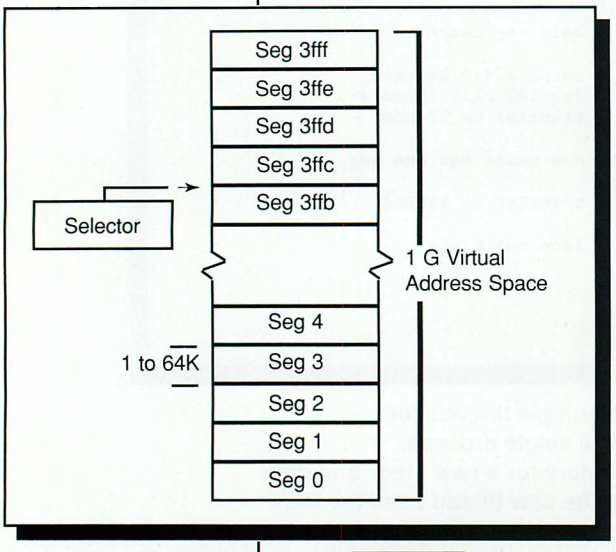
```
stksiz    equ   1024                ; size of new thread's stack
                                    :
Selector  dw    ?                   ; selector from DOSALLOCSEG

BeeperID  dw    ?                   ;  Thread ID for new thread
                                    ; named 'beeper'
                                    :
          push  stksiz              ; size of new segment
          push  ds                  ; address of variable
          push  offset DGROUP:Selector  ; to receive new selector
          push  0                   ; non-shared segment
          call  DOSALLOCSEG         ; TRANSFER TO 286DOS
          or    ax,ax
          jnz   error               ; jump if alloc failed

          push  cs                  ; execution address of
          push  offset_TEXT:Beeper  ; new thread
          push  ds                  ; address to receive new
          push  offsetDGROUP;BeeperID  ; Thread ID
          push  Selector            ; address of base of
          push  stksiz              ; new thread's Stack
          call  DOSCREATETHREAD     ; transfer to 286DOS
          or    ax,ax
          jnz   error               ; jump if create failed
                                    :
beeper    proc  far                 ; entry point for thread

          mov   cx,10               ; emit ten beeps...

beep1:    push  440                 ; sound a 440 Hz tone
          push  100                 ; for 100 milliseconds
          call  DOSBEEP             ; transfer to 286DOS

          push 0                    ; now pause for one sec.
          push  1000
          call  DOSSLEEP            ; transfer to 286DOS

          loop  beep1               ; loop ten times
          ret

beeper    endp
                                    :
```

▲ **Figure 4** **An example of the use of multiple threads for asynchronous execution of tasks within a single process. The main line of execution allocates memory for a new stack and then starts up another thread called beeper. The new thread uses the OS/2 service DOSBEEP to emit ten short tones at one-second intervals and then terminates.**

OS/2 Memory Management



▲ **Figure 1** **Real Address Mode Segment Selector Interpretation. The selector is used to identify segments in real memory.**



▲ **Figure 2** **In protected mode the selector does not specify a segment's location in physical memory. Instead, it uniquely identifies (or names) any one of 16K possible segments—the selector uses 14 bits to address the segments in a given task's virtual address space. Because each segment can address 64K, the total virtual space available is 1Gb**

In a multitasking system, random access memory (RAM) must be administered as a critical, finite system resource akin to disk storage or a line printer. The well-bred application program uses the host operating system's facilities to request additional memory politely when necessary, releases unneeded memory, and abstains from meddling with memory that doesn't belong to it.

In the MS-DOS environment, such genteel programs are few and far between. Most popular application programs place the goal of high performance above all else, and a disposition to grab all available memory, capture interrupt vectors, and directly access the video controller's refresh buffer for fast screen displays is the rule rather than the exception. This behavior causes many headaches for those system software vendors who seek to graft multitasking abilities onto MS-DOS with such new user interfaces as *Windows*, *TopView*, and *DESQview*.

The 80286 microprocessor, which is the heart of the IBM PC AT and compatible computers, can execute in either of two modes: real mode and protected mode. When the 80286 is running in real mode, as it does under MS-DOS, it essentially functions as a fast 8086/88 processor with a few added machine instructions. The segment registers contain physical addresses, which can be manipulated directly, and the total amount of memory that can be addressed is 1 megabyte (Mb) (see **Figure 1**). On the 8086, and the 80286 running in real mode, there is no way for the operating system to intervene when a program does not use memory properly or to prevent one program from writing into another's memory space.

Under OS/2, however, the rules are changed. OS/2 runs the 80286 processor in protected mode, providing hardware support for memory protection and memory access privilege levels, and presenting a very different hardware architecture to the programmer. The contents of segment registers are now logical selectors rather than physical addresses (see **Figure 2**). Selectors are indexes into descriptor tables that contain the actual physical address and length of the corresponding chunk of memory, and the descriptor tables themselves are typically not accessible to application programs at their privilege level. Each segment has an attribute that determines how it can be accessed: executable, read-only data, or read-write data.

Memory protection between programs is enforced by each process's local descriptor table (LDT), a map of the memory segments that can be accessed by that process. If a task tries to load a segment register with a selector that is not valid for its own descriptor table or for the global descriptor table (GDT), a hardware interrupt is generated (see **Figure 3**). The operating system then recovers control, terminates the offending process, and displays di-

agnostic information on the system console.

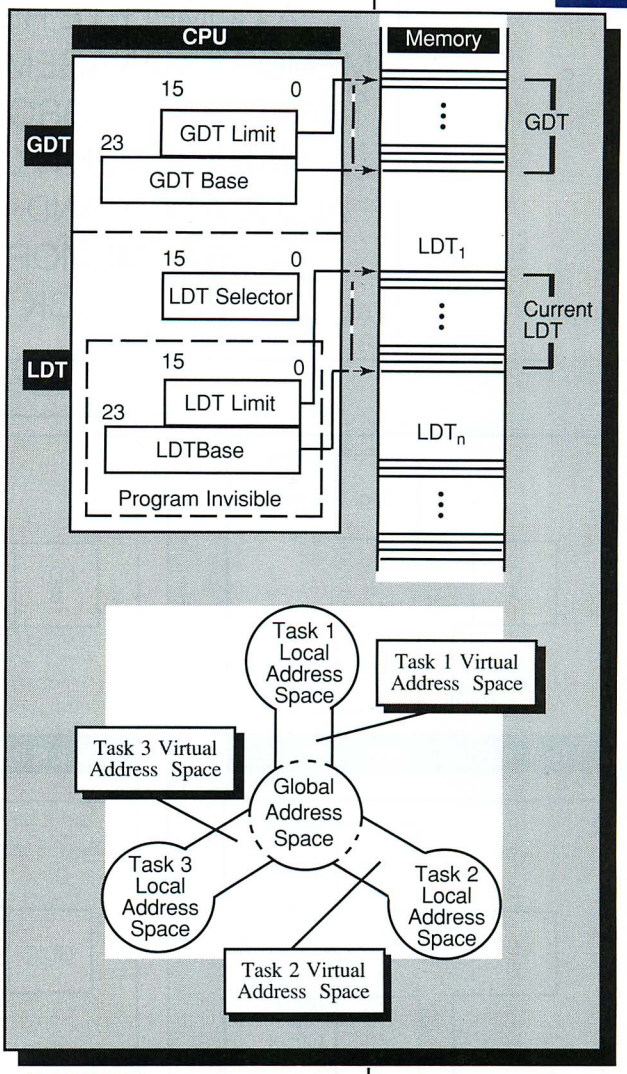
Thus, applications running under OS/2 must be civil to survive: there is no alternative to using the host system's services for managing memory. Only the operating system is entrusted with the privilege level and the knowledge of physical memory layout, which is necessary to create and destroy memory segments through manipulation of descriptor tables. Fortunately, OS/2 provides applications with a complete repertoire of memory management services, in three major groups: local heap management, conventional allocation and release of global memory segments, and "huge" global block management (logically contiguous memory spaces that are larger than 64Kb).

In exchange for good behavior, the 80286's protected mode provides a reward—virtual memory (see **Figure 4**). A task can request, and own, more bytes of RAM memory than physically exist in the system. The combination of hardware support for *paging* and a specialized module of the operating system called the *swapper* present to the task the illusion that all of its segments are simultaneously present and accessible.

When all of physical memory is in use and a process attempts to allocate additional memory, the least recently used segment is copied to a *swap file* on the disk, and the memory it occupied is freed up. A special bit in the descriptor table entry for the swapped-out segment is set to indicate that it is nonresident. When a selector for a swapped-out segment is used in a memory access, a hardware interrupt called a *page fault* occurs. The virtual memory manager (swapper) gains control, reads the needed segment from the disk into physical memory (possibly writing another segment out first to make room), updates the descriptor table entry for the selector to correspond to the new physical address, and restarts the process's instruction that was trying to access the segment.

Thus, virtual memory means that the presence or absence in physical memory of any particular segment is completely transparent to the process that owns it. If an address that is not present is accessed, it will be invisibly loaded on demand. Correspondingly, the amount of memory that can be committed by all the processes in the system combined is limited only by the amount of swapping space on the disk (although limits on the number of available segment selectors do exist, they are of no practical significance owing to the sizes of today's fixed disks).

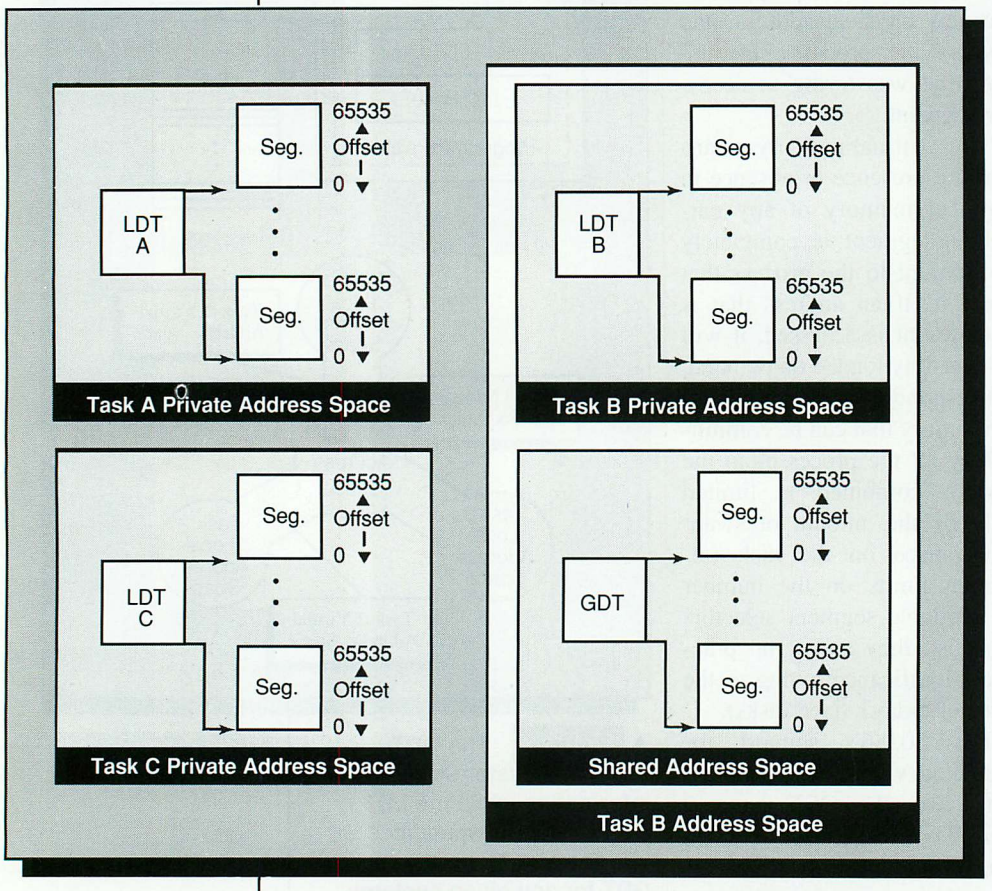
The 80286's support for protected, virtual memory facilitates two other OS/2 features: *shared text* (code) segments and *dynamic linking*. Segments containing machine instructions are marked with the executable attribute and cannot be modified. Thus, when the same program is loaded for execution in two or more sessions, each *instantiation* can share the same memory-resident piece of code, instead of requiring separate, distinct copies. The protected-mode



▲ **Figure 3** **The GDT maintains system-wide information while the LDT maintains task-specific information. There is one GDT for any given system that is shared by every task or process. Each task's virtual local space, managed by the LDT, is isolated from that of any other. OS/2 itself resolves virtual addresses to map tasks to real memory.**

Figures reprinted with permission of Intel Corporation.

OS/2 PROVIDES APPLICATIONS WITH A COMPLETE REPERTOIRE OF MEMORY MANAGEMENT SERVICES, IN THREE MAJOR GROUPS: LOCAL HEAP MANAGEMENT, CONVENTIONAL ALLOCATION AND RELEASE OF THE GLOBAL MEMORY, AND "HUGE" GLOBAL BLOCK MANAGEMENT.



▲ **Figure 4** 80286 Virtual Address Space. OS/2 provides one GDT that is shared by each task or process. Hardware support for paging and the OS/2 swapper together provide the illusion that all segments are simultaneously present and accessible.

command processor called CMD.EXE is a good example: the overhead of multiple screen groups is small, since only one copy of the machine code in CMD.EXE is needed. Adding another screen group with its own command processor requires only the creation of another data segment and virtual screen buffer.

Dynamic link libraries are simple extensions of the concepts of virtual memory and shared text segments. Under MS-DOS, when a program was linked to a run-time library, the actual machine code for the library routines became a permanent part of the application. Under OS/2, the linker and the .EXE file header have been extended to allow *run-time binding* (**Figure 5**) of library routines. When a program containing dynamic links is loaded for execution, OS/2 examines the file header and loads any modules from dynamic link libraries that are required, then updates the FAR CALL addresses within the application. Naturally, these dynamically linked routines can be shared between multiple processes and will be discarded and reloaded as needed by the memory manager. Much of OS/2 itself is implemented in the form of dynamic link libraries. ❑

DOSLOADMODULE	Load Dynamic Link Module
DOSFREEMODULE	Free Dynamic Link Module
DOSGETPROCADDR	Get Dynamic Link Procedure Address
DOSGETMODHANDLE	Get Dynamic Link Module Handle
DOSGETMODNAME	Get Dynamic Link Module Name

▲ **Figure 5** To invoke *run-time dynamic linking*, the application uses the following system calls.

OS/2 Inter-Process Communication:Semaphores, Pipes, and Queues

Ray Duncan

Operating System/2 multitasking services allow applications to create multiple concurrent threads within a process or to create child processes. However, as we have seen earlier (see "OS/2 Multitasking: Exploiting the Protected Mode of the 80286," page 27), the multitasking functions provide a parent process with only a very limited ability to influence and communicate with a child process. The parent process can pass information to the child at load time in the form of command strings and the environment block, it can obtain the child process's exit (return) code and a code describing the child process's method of termination, and unilaterally abort a child process.

To fill the need for rapid exchange of all kinds of information between concurrent threads and related or unrelated processes, OS/2 includes a rich set of system services called inter-process communication (IPC) functions. Computer literature is teeming with different mechanisms for inter-process communication, and more are being added every day. Still, there is broad agreement on three basic methods of IPC, all of which are supported by OS/2: semaphores, pipes, and queues.

In this article, we'll briefly discuss each of these three IPC mechanisms, summarize the relevant OS/2 functions, and supply a brief coding example. Since there are many options available in the IPC services, we can't provide a comprehensive description of their capabilities here; the program fragments are only intended to demonstrate the most common usage. More-detailed information can be found in the *Software Development Kit* (see "The Microsoft OS/2 Software Development Kit," page 60) and in the OS/2 *Programmer's Reference* manual.

While reading about the IPC functions, bear in mind the distinctions between processes and threads. When a process opens or creates a semaphore, pipe, or queue, a handle is returned that can be used by any thread within that process and is inherited by any child processes. On the other hand, when a thread issues an OS/2 function call that waits on a semaphore or performs a synchronous read or write to a pipe or queue, only the calling thread is suspended—other threads in the process continue to run as they did before. When

TO FILL THE NEED FOR RAPID EXCHANGE OF ALL KINDS OF INFORMATION BETWEEN CONCURRENT THREADS AND RELATED OR UNRELATED PROCESSES, MICROSOFT OPERATING SYSTEM/2 INCLUDES A RICH SET OF SYSTEM SERVICES CALLED INTER-PROCESS COMMUNICATION (IPC) FUNCTIONS. OS/2 SUPPORTS SEMAPHORES, PIPES, AND QUEUES.

Ray Duncan is a regular columnist for Dr. Dobb's Journal of Software Tools *and author of the best-selling* Advanced MS-DOS, *published by Microsoft Press.*

Access to system semaphores	
DOSCREATESEM	Create a system semaphore
DOSOPENSEM	Open an existing system semaphore
DOSCLOSESEM	Close a system semaphore
Semaphore functions for resource control	
DOSSEMREQUEST	Establish ownership of a semaphore
DOSSEMCLEAR	Release ownership of a semaphore
Semaphore functions for signaling	
DOSSEMSET	Unconditionally set a semaphore
DOSSEMCLEAR	Unconditionally clear a semaphore
DOSSEMWAIT	Wait until semaphore is cleared
DOSSEMSETWAIT	Unconditionally set semaphore and wait until it is cleared by another thread
DOSMUXSEMWAIT	Wait for any of several semaphores to be cleared

▲ **Figure 1** **A summary of OS/2 semaphore support functions. Semaphores can be used to coordinate access to a resource or for signaling between threads or processes.**

```
        push 1              ; exclusive ownership not desired
        push ds             ; variable to receive semaphore handle
        push offset DGROUP:shandle
        push ds             ; address of semaphore name
        push offset DGROUP:sname
        call DOSCREATESEM ; transfer to OS/2
        or   ax,ax          ; create successful?
        jz   continue       ; jump if it was...
        cmp  ax,err_sem_exists    ; semaphore already exists?
        jnz  error          ; jump if some other error

                            ; semaphore exists, open it instead
        push ds             ; variable to receive handle
        push offset DGROUP:shandle
        push ds             ; address of semaphore name
        push offset DGROUP:sname
        call DOSOPENSEM     ; transfer to OS/2
        or   ax,ax          ; was open successful?
        jnz  error          ; if can't open and can't create,
                            ; something is terribly wrong
                                  :
sname   db   '\SEM\RESOURCE.LCK',0

shandle dd   0
```

▲ **Figure 2** **A code fragment that illustrates the creation or opening of a system semaphore.**

multiple threads in the same process issue requests against the same IPC object, any necessary serialization of these requests is taken care of within the operating system.

Semaphores

A semaphore can be viewed as a simple object with two states: set (owned) or cleared (not-owned). OS/2 provides a complete battery of system services to test, set, and clear semaphores; a summary of these functions is found in **Figure 1**. Semaphores are a high-performance mechanism of inter-process communication, because they are always resident in memory and the OS/2 services that manipulate them are relatively fast.

The classical use of semaphores, and a usage fully supported by OS/2, is to control access to a nonreentrant routine or a serially reusable resource (SRR). An SRR is a file, peripheral device, or a memory object that is damaged or produces unpredictable results if it is accessed by more than one thread or process at a time. With semaphores, mutual exclusion on such a resource is easy to arrange between cooperating processes. A semaphore is established that represents the resource, and a thread or process refrains from accessing the resource unless it "owns" the corresponding semaphore.

An alternative use of semaphores in OS/2 is to provide for synchronization or signaling between threads or processes. In this usage, one or more threads can suspend themselves by using an OS/2 function to "block" or wait on a semaphore. When the semaphore is cleared by another thread in response to some event, all of the threads that were blocking on that semaphore will wake up and run. When a semaphore is used for signaling and no resource that can be corrupted is involved, any thread that knows about the semaphore can set, clear, or test the semaphore at any time.

To provide for the slightly different requirements of inter-

thread and inter-process communication, OS/2 supports two types of semaphores: RAM semaphores and system semaphores.

RAM semaphores are used for signaling or resource control between multiple threads in a single process. Each RAM semaphore requires the allocation of a double word of data storage; the double word should be initialized to zero when the process is started. The handle for a RAM semaphore is simply its 32-bit address (selector and offset).

System semaphores are named objects that are used for signaling or synchronization between processes; the name always takes the form:

```
\SEM\name.ext
```

The extension (.ext) is optional. The storage for a system semaphore is allocated by the operating system somewhere outside the creating processes' memory space.

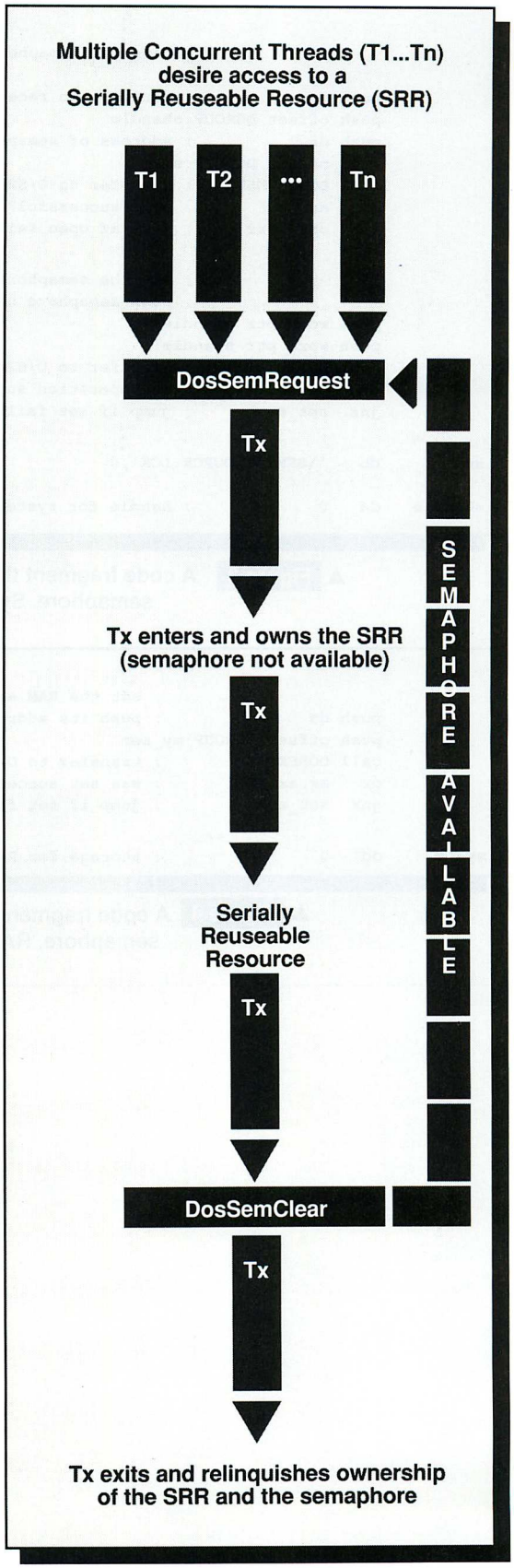
Access to a system semaphore is obtained by creating or opening it by name; OS/2 returns a 32-bit handle that can be used for subsequent access to the semaphore. When the process no longer requires access to the system semaphore, it can use the handle to close it as it would close a file. The three functions that control access to a system semaphore, DOSCREATESEM, DOSOPENSEM, and DOSCLOSESEM, are described in more detail below.

Creating a system semaphore. The function DOSCREATESEM is called with the address of a null-terminated (ASCIIZ) semaphore name, the address of a double-word variable that will receive the semaphore handle, and a flag that indicates whether ownership of the semaphore will be "exclusive"—in which case the state of the semaphore cannot be altered by a process that does not own it (see **Figure 2**). If the create operation is successful, the initial state of the semaphore will be not-owned (cleared).

Opening a system semaphore. The function DOSOPENSEM is called with the address of a null-terminated (ASCIIZ) semaphore name of an existing semaphore and the address of a double word of storage that will receive a handle. Opening a semaphore does not establish ownership, test, or change the value of the semaphore.

When a process creates or opens system semaphores, the handles to those semaphores are inherited by any child processes that are started with DOSEXECPGM. However, the child process does not own the semaphores even if the parent owned them; only one process at a time can own a given semaphore.

Closing a system semaphore. The function DOSCLOSESEM is called with a



RAM Semaphores: A thread (Tx) issues the DosSemRequest call to claim access to the semaphore controlling a given SRR. If the semaphore is currently unused, DosSemRequest gives the calling thread ownership of the semaphore. All other threads are locked out until the owning thread clears the semaphore through DosSemClear and relinquishes ownership of both the SRR and the semaphore.

```
                                    :
                                    :
                            ; open the semaphore

        push ds             ; variable to receive handle
        push offset DGROUP:shandle
        push ds             ; address of semaphore name
        push offset DGROUP:sname
        call DOSOPENSEM     ; transfer to O/S2
        or   ax,ax          ; open successful?
        jnz  open_err       ; jump if open failed
                                    :
                                    :
                            ; set the semaphore:
                            ; push semaphore handle...
        push word ptr shandle+2
        push word ptr shandle
        call DOSSEMSET      ; transfer to O/S2
        or   ax,ax          ; was operation successful?
        jnz  set_err        ; jump if set failed
                                    :
                                    :
sname   db   '\SEM\RESOURCE.LCK',0

shandle dd   0              ; handle for system semaphore
```

▲ **Figure 3** **A code fragment that demonstrates setting a system semaphore. System semaphores are best suited for use "between" processes.**

```
                                    :
                                    :
                                ; set the RAM semaphore
        push ds                 ; push its address
        push offset DGROUP:my_sem
        call DOSSETSEM          ; transfer to O/S2
        or   ax,ax              ; was set successful?
        jnz  set_err            ; jump if set failed
                                    :
                                    :
mysem   dd   0                  ; storage for RAM semaphore
```

▲ **Figure 4** **A code fragment that demonstrates setting a RAM semaphore. RAM semaphores are best suited for use "within" processes.**

handle and terminates access to a system semaphore. If a process terminates with open semaphores, they are closed automatically.

However, if any of these semaphores were owned by the process, any threads in other processes that were waiting on the semaphore are awakened and receive an error code that indicates that the owning process may have terminated abnormally and the resource controlled by the semaphore may be in an indeterminate state. The semaphore itself ceases to exist when processes that use the semaphore have terminated or called DOSCLOSESEM.

Mutual Exclusion

The two OS/2 functions that are associated with the classical use of a semaphore are DOSSEMREQUEST and DOSSEMCLEAR.

Claiming a semaphore. The OS/2 function DOSSEMREQUEST is used by a thread to establish ownership of a semaphore and, by inference, ownership of the resource associated with the semaphore. DOSSEMREQUEST is called with the handle of a system or RAM semaphore and a time-out parameter. If the specified semaphore is currently unowned, the function sets it as owned and returns a success code. If the semaphore is already owned by another thread, the system either suspends the thread indefinitely until the semaphore becomes available, waits for a specified interval and then returns with an error code, or returns immediately with an error code, depending on the time-out parameter.

Recursive requests for system semaphores are supported by means of a use count, maintained by OS/2, of the number of times the semaphore owner has issued DOSSEMREQUEST without a corresponding DOSSEMCLEAR. However, recursive use of RAM semaphores is not supported.

Releasing a semaphore. The function DOSSEMCLEAR is called with a semaphore handle and releases the thread's ownership of that semaphore. An error code is returned if the handle is invalid or if the semaphore was created with the exclusive option and is not currently owned by the calling thread. If the semaphore is already cleared, then no error is returned.

When multiple threads are waiting on the same semaphore with DOSSEMREQUEST, it can be difficult to predict which thread will be awakened and acquire the semaphore when it is released by the current owner. The thread selected depends on an interaction of the priorities of the waiting threads, the position of the threads in the scheduler's list, and the activity of other threads in the system.

Signaling

Semaphores are used as signals when only one thread is in a position to detect an event but other threads may wish to take action based on this event. For example, a thread in a detached process utility that is monitoring the keyboard data stream for a hot key might wish to trigger other threads in the same process to display a pop-up screen or write to a file. The MS OS/2 functions that support signaling include DOSSEMSET, DOSSEMCLEAR, DOSSEMWAIT, DOSSEMSETWAIT, and DOSMUXSEMWAIT.

Setting a semaphore. The OS/2 function DOSSEMSET is called with the handle of a system or RAM semaphore and unconditionally sets that semaphore (see **Figures 3** and **4**). The function fails if the handle is invalid or if a system semaphore was created with the exclusive option and is currently owned by another process (a semaphore used for signaling would not ordinarily be created with the exclusive option).

Clearing a semaphore. The function DOSSEMCLEAR is called with the handle of a RAM or system semaphore and unconditionally clears that semaphore. As with DOSSEMSET, the function fails if the semaphore handle is invalid or if the semaphore was created with the exclusive option and is currently owned by another process. If any threads were blocked on the semaphore, they are restarted.

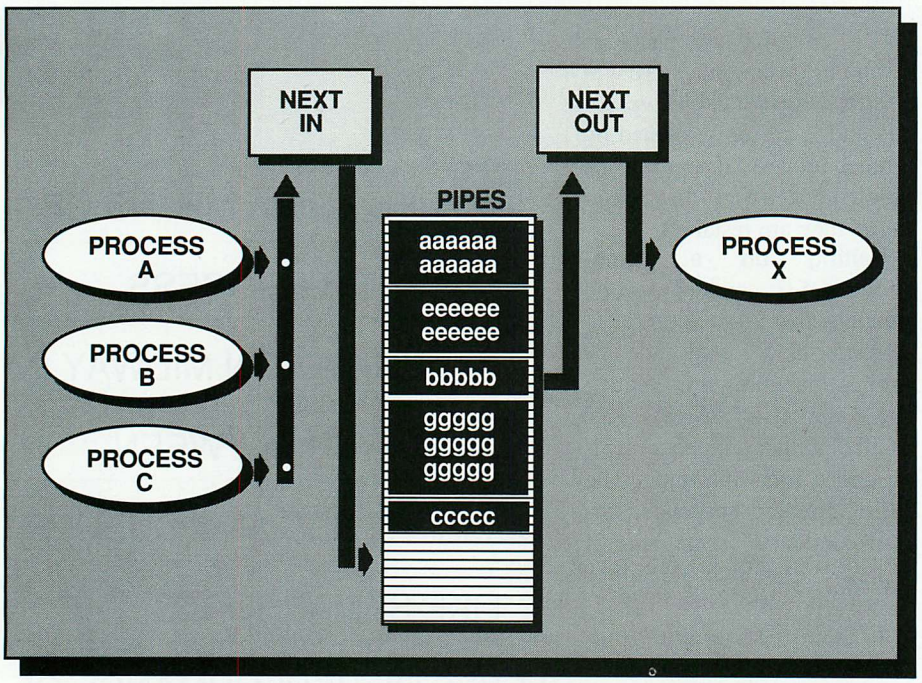
Waiting on a semaphore. OS/2 contains several functions that allow a thread to suspend itself until one or more semaphores are cleared. The function DOSSEMWAIT is called with a semaphore handle and a time-out value. The calling thread is suspended until the indicated semaphore is cleared by another thread or process with DOSSEMCLEAR; it regains control immediately if the semaphore is not set. The function returns an error if the handle is invalid, no-wait was specified, and the semaphore is currently set; if a finite wait was specified and timed out; or if the semaphore was created with the exclusive option and is currently owned by another process.

The function DOSSEMSETWAIT works just like DOSSEMWAIT, except that it sets the semaphore if it was not already set and then suspends the current thread until the semaphore is cleared by another thread or process or the indicated time-out expires.

DOSSEMWAIT and DOSSEMSETWAIT are level-triggered, not edge-triggered. This means that it is possible for a thread that is blocking on a semaphore to miss a quick clearing and resetting of the semaphore, depending on the thread's priority and position in the scheduler's list and the activity of the other threads in the system.

The function DOSMUXSEMWAIT is called with a list of up to 16 semaphores and an

PIPES ARE A MEANS OF INTER-PROCESS COMMUNICATION MIDWAY IN POWER BETWEEN SEMAPHORES AND QUEUES. LIKE SEMAPHORES, THE LEVEL OF PERFORMANCE OF PIPES IS RELATIVELY HIGH, BECAUSE THE INFORMATION IS ALWAYS KEPT RESIDENT IN MEMORY; LIKE QUEUES, PIPES CAN BE USED TO PASS ANY TYPE OF DATA BETWEEN PROCESSES.



Communicating with a Pipe: Pipes are a high performance mechanism for communication between closely related processes (such as a parent process and multiple child processes). Pipes are read and written like a file, and may contain variable-length messages with any content.

```
                                    :
                                    :
        push ds                 ; address to receive handle
        push offset DGROUP:prdh       ; for reading pipe
        push ds                 ; address to receive handle
        push offset DGROUP:pwriteh    ; for writing pipe
        push 0                  ; max pipe size = default
        call DOSMAKEPIPE        ; transfer to O/S2
        or   ax,ax              ; was pipe created?
        jnz  error              ; jump if create failed.
                                    :
                                    :
                                ; in here we spawn a "child"
                                ; process which inherits handles...

                                ; now send a message to child
                                ; process through the pipe...


        push pwriteh            ; pipe write handle
        push ds                 ; address of message
        push offset DGROUP:message
        push message_length     ; length of message
        push ds                 ; address of variable to
        push offset DGROUP:status     ; receive bytes written
        call DOSWRITE           ; transfer to O/S2
        or   ax,ax              ; did write succeed?
        jnz  error              ; jump if write failed.
                                    :
                                    :
preadh  dw   ?                  : handle to read pipe

pwriteh dw   ?                  ; handle to write pipe

status  dw   ?                  ; receives length from DOSWRITE
```

optional time-out interval. The calling thread is suspended until any of the indicated semaphores are cleared or the indicated interval has elapsed. Unlike DOSSEMWAIT and DOSSEMSETWAIT, the function DOSMUXSEMWAIT is edge-triggered; the waiting thread is awakened when one of the semaphores in the list changes state even if that semaphore gets set (armed) again right away by another process or another thread.

Pipes

Pipes are a means of interprocess communication midway in power between semaphores and queues. Like semaphores, the level of performance of pipes is relatively high, because the information is always kept resident in memory; like queues, pipes can be used to pass any type of data between processes. Physically, a pipe is simply a chunk of memory that is used as a ring buffer, with In and Out pointers maintained by the system. From a process's point of view, reading and writing a pipe are equivalent to reading and writing a file, except that a pipe is much faster.

Pipes are not named entities. A process creates a pipe by calling the OS/2 function DOSMAKEPIPE with the addresses of two variables to receive the read and write handles for the pipe, and a maximum pipe size (up to 65,504 bytes). The function fails if there is not enough memory to create the pipe or if no handles are available.

The two handles for reading and writing the pipe are as-

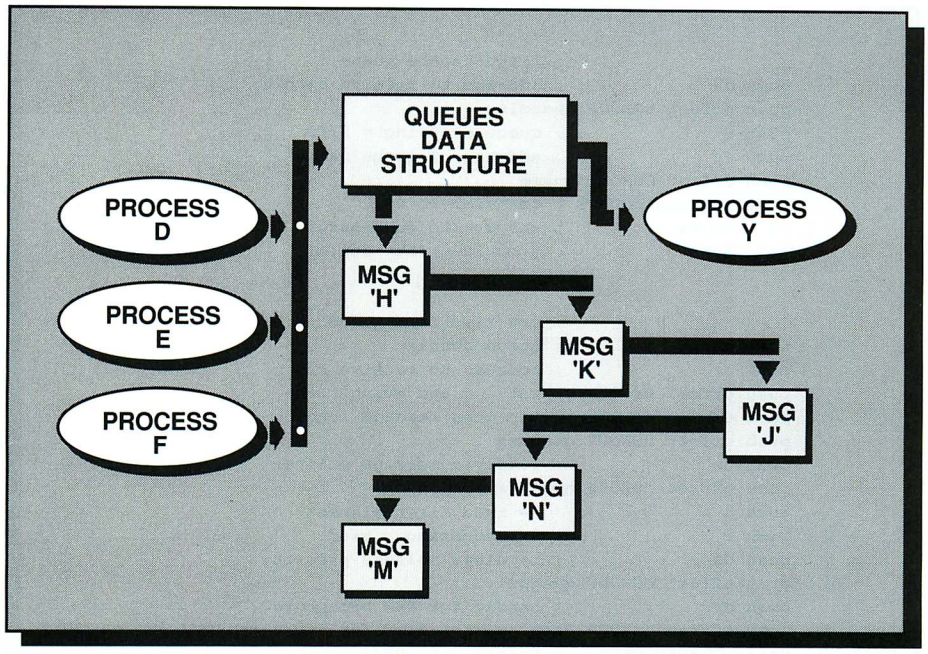
◀ **Figure 5**
Establishing Pipes

signed out of the same numeric sequence as those used for access to files. Any child processes automatically inherit the handles and have access to the pipe. We have already encountered the two major restrictions on the use of pipes: they can only be used for communication between closely related processes, and the maximum amount of data that a pipe can contain at one time is relatively small.

Threads read and write a pipe with the usual file DOSREAD and DOSWRITE calls, supplying a handle, buffer address, and record length (see **Figure 5**). If a thread attempts to write to a pipe and the pipe is full, that thread is suspended until another thread or process removes enough data from the pipe so that the write can be completed. If a thread requests a synchronous read from a pipe and the pipe is empty, that thread is suspended until some other thread writes enough data to the pipe to satisfy the read request. There are no control or permission mechanisms or checks performed on operations to pipes.

A pipe vanishes from the system when all the processes using the pipe either close the pipe handles or terminate. If two processes are communicating with pipes and the process reading the pipe ends, the process writing the pipe receives an error code.

The most common usage of pipes by a process—and the way that the system's command interpreter uses them—is to arrange for the pipe's handles to be substituted for the standard input and standard output handles. Any child processes that are started then automatically communicate with the par-



Communicating with a Queue: Queues are the most flexible and powerful mechanism of inter-process communication supported by OS/2. A queue is basically an ordered list of shared memory segments; each segment contains a separate message and may be as large as 64K. The messages may be ordered in the queue by First-In-First-Out, Last-In-First-Out, or priority, and may be inspected or removed from the queue by the server process in any order whatsoever.

Queue access	
DOSCREATEQUEUE	Create a queue (process is owner, can read or write messages to queue)
DOSOPENQUEUE	Open a previously existing queue (can only write messages to queue)
DOSCLOSEQUEUE	Close queue (queue is destroyed when closed by owner)
Queue input/output	
DOSWRITEQUEUE	Write message to queue (either queue owner or any process that has opened queue)
DOSREADQUEUE	Read message from queue (queue owner only)
DOSPEEKQUEUE	Nondestructive read of queue message (queue owner only)
Queue information	
DOSQUERYQUEUE	Find number of messages currently in queue (queue owner or any process that has opened queue)
DOSPURGEQUEUE	Discard all messages currently in queue (queue owner only)

▲ **Figure 6** **Substantial control over queues is provided by the OS/2 queue management functions.**

```
                                    :
                                    :
                            ; first create queue...
        push ds             ; address to receive handle
        push offset DGROUP:qhandle
        push 0              ; queue ordering = FIFO
        push ds             ; address of queue name
        push offset DGROUP:qname
        call DOSCREATEQUEUE ; transfer to OS/2
        or   ax,ax          ; was create successful?
        jnz  error          ; jump if create failed


                            ; now read from queue
        push qhandle        ; queue handle
        push ds             ; address to receive PID
        push offset DGROUP:qident   ; and event code
        push ds             ; receives message length
        push offset DGROUP:qmsglen
        push ds             ; receives message pointer
        push offset DGROUP:qmsgptr
        push 0              ; 0= read first element
        push 0              ; 0= synchronous read
        push ds             ; receives message priority
        push offset DGROUP:qmsgpri
        push ds             ; handle for RAM semaphore
        push offset DGROUP:qsem  ; (not used for synch reads)
        call DOSREADQUEUE   ; transfer to OS/2
        or   ax,ax          ; was read successful?
        jnz  error          ; jump if read failed

        les  bx,qmsgptr     ; let ES:BX point to queue msg


        push es             ; now release queue message's
        call DOSFREESEG     ; shared memory segment
        or   ax,ax          ; was release successful?
        jnz  error          ; jump if released failed


        push qhandle        ; now close the queue
        call DOSCLOSEQUEUE  ; also destroying it
        or   ax,ax          ; was close successful?
        jnz  error          ; jump if close failed
                                    :
                                    :
qhandle dw   ?              ; receives handle from DosCreateQueue

qname   db   '\QUEUES\MYQUEUE',0 ; ASCIIZ queue name

qident  dw   0,0            ; writer's PID, event code

qmsglen dw   ?              ; length of queue msg received

qmsgptr dd   0              ; address of queue msg received

qmsgpri dw   ?              ; priority of queue msg
```

▲ **Figure 7** Creating a queue, waiting for a message to be written into it, and then closing the queue (destroying it).

ent process for their input and output instead of the keyboard and screen, without any special knowledge or action on the child's part.

Queues

Queues are the most powerful inter-process communication service provided by OS/2, and consequently the most complex to use. Queues are slower than pipes, and much slower than semaphores, but they are also far more flexible.

- Queues are named objects and can be opened and written by any process.
- The size of a queue is limited only by the amount of free memory plus swap space on the disk.
- Each record in a queue is a separately allocated block of memory storage and can be as large as 65,536 bytes.
- The records in a queue may be ordered by the first-in-first-out (FIFO), last-in-first-out (LIFO), or priority methods.
- The queue owner can examine records in the queue and remove them selectively, in any order.
- Data in the queue is not copied from place to place by the operating system; instead, the data is passed in shared memory segments.

The name of a queue will always take the form of \QUEUES\name.ext where the extension (.ext) is optional. When a queue is created or opened, OS/2 returns a handle that is used for subsequent access to the queue. A summary of OS/2 queue services can be found in **Figure 6**.

Creating a queue. The function DOSCREATEQUEUE is used to originate a new queue in the system. The function is called with a null-

terminated (ASCIIZ) string that names the queue, a parameter that specifies the queue ordering (FIFO, LIFO, or priority), and the address of a variable to receive the queue handle (see **Figure 7**). An error code is returned if a queue already exists with the same name, the given name is invalid, or there is not enough free memory in the system to establish the queue's supporting data structure. Only the process that creates the queue, called the queue owner, can remove records from the queue, but all threads in the owning process can access the queue with equal authority.

Opening a queue. Any process can open an existing queue by calling the OS/2 function DOSOPENQUEUE with the name of the queue and the address of two variables to receive the queue handle and the PID of the queue owner. The handle is used for subsequent writes of records to the queue.

Writing a queue. Adding records to a queue is much more involved than writing a record to a pipe. The writer must first allocate a chunk of global shared memory of appropriate size with the function DOSALLOCSEG and copy the data for the queue record into it. Next, the writer must use the function DOSGIVESEG to obtain a new selector for the memory segment that can be passed to another process. Finally, the writer calls DOSWRITEQUEUE with a queue handle, a priority, the address (from DOSGIVESEG) and length of the data to be added to the

```
                                        :
                                        :
        push ds         ; address to receive PID of
        push offset DGROUP:qowner       ; queue's owner
        push ds         ; address to receive handle
        push offset DGROUP:qhandle
        push ds         ; address of queue's name
        push offset DGROUP:qname
        call DOSOPENQUEUE       ; transfer to OS/2
        or   ax,ax              ; was open successful?
        jnz  error              ; jump if open failed


        push qmsglen            ; length of shared memory segment
        push ds                 ; receives selector for segment
        push offset DGROUP:qselc
        push 1                  ; 1 = segment will be shared
        call DOSALLOCSEG        ; transfer to OS/2
        or   ax,ax              ; was allocation successful?
        jnz  error              ; jump if memory allocate failed


        mov  si,offset qmsg     ; copy our queue message
        mov  es,qselc           ; to shared segment
        xor  di,di
        mov  cx,qmsglen
        cld
        rep movsb


        push qselc              ; get a new selector for the
        push qowner             ; shared segment that can
        push ds                 ; be used by receiver of
        push qselr              ; the queue message
        call DOSGIVESEG         ; transfer to OS/2
        or   ax,ax              ; did function succeed?
        jnz  error              ; jump if can't give it away


        push qhandle            ; handle for queue
        push 0                  ; 0= event data
        push qmsglen            ; length of queue message
        push qselr              ; selector for queue data
        push 0                  ; (offset)
        push 0                  ; queue element priority
        call DOSWRITEQUEUE      ; transfer to OS/2
        or   ax,ax              ; did write succeed?
        jnz  error              ; jump if queue write failed


        push qselc              ; release shared memory that
        call DOSFREESEG         ; contains the queue message
        or   ax,ax              ; did release succeed?
        jnz  error              ; jump if release failed


        push qhandle            ; now close the queue
        call DOSCLOSEQUEUE      ; transfer to OS/2
        or   ax,ax              ; did close succeed?
        jnz  error              ; jump if close failed
                                        :
                                        :
qhandle dw   ?          ; receives handle from DosOpenQueue

qowner  dw   ?          ; receives PID of queue owner

qname   db   '\QUEUES\MYQUEUE',0 ; ASCIIZ queue name

qselc   dw   ?          ; shared memory selector for caller

qselr   dw   ?          ; shared memory selector for receiver

qmsg    db   'This is my queue message'
qmsglen equ  $-qmsg
```

Figure 8 ▶
Opening a queue, writing a message into it, and then closing it.

THE SELECTION OF A PARTICULAR OS/2 INTER-PROCESS COMMUNICATION TECHNIQUE FOR A GIVEN APPLICATION MUST BE MADE CAREFULLY ON THE BASIS OF PERFORMANCE REQUIREMENTS AND THE CHARACTER OF THE MESSAGES TO BE EXCHANGED BETWEEN THE INTERESTED THREADS OR PROCESSES.

queue (see **Figure 8**). The queue write proper fails if the queue handle is invalid or if there is insufficient system memory to expand the supporting structure of the queue. Obviously, the sequence can also fail at an earlier point (in DOSALLOCSEG or DOSGIVESEG) if the system runs out of memory (the physical memory plus swap space on the disk is exhausted) or selectors.

Reading a queue. The function DOSREADQUEUE is used by the queue owner to remove a record from a queue; it is called with a queue handle and several other selection parameters and returns the address of a queue element (selector and offset) and its length. The owner can choose a record from the queue based on its position or priority or simply take the next record in line. The read operation can also be synchronous (the calling thread is suspended until a record is available) or asynchronous (the calling thread gets control back immediately, and a semaphore is set when a record is available).

The function DOSPEEKQUEUE works similarly to DOSREADQUEUE, but the record is not removed from the queue when it is retrieved. This allows the queue owner to scan through the data waiting in the queue and decide on a processing strategy, without disturbing the data structure or being forced to copy the records to its own memory.

Miscellaneous operations. The function DOSQUERYQUEUE allows any process that has a valid handle for a queue to obtain the current number of elements in that queue. DOSPURGEQUEUE discards all records that are currently in the queue; this function can only be called by the queue owner.

Closing a queue. The function DOSCLOSEQUEUE is called with a queue handle and informs OS/2 that the calling process will not require further access to the queue. If the closing process is also the queue owner, the queue is purged and destroyed—any further attempts to write records to the queue by other processes will fail.

Summary

OS/2 supports three methods of inter-process communication that are used to pass messages between threads or processes: semaphores, pipes, and queues.

Semaphores are used to symbolize ownership of a resource or signal occurrence of an event; they can be thought of as simple flags that can be set, cleared, or tested.

Pipes are a high-performance means of passing variable-length, variable-content data between two closely related processes; however, pipes have the relative disadvantage that their maximum size is fixed at the time that they are created and they can never hold more than 64K.

Queues allow a vast amount of prioritized data to be passed from multiple client processes to a server process; each message in the queue can be as large as 64K, and the total amount of data in the queue is limited only by the system's virtual memory. The selection of a particular IPC technique for a given application must be made carefully on the basis of performance requirements and the character of the messages to be exchanged between the interested threads or processes. ❑

The MS OS/2 LAN Manager

System connectivity has become an increasingly significant issue in the development of today's personal computing systems. As a result, it is becoming substantially more important for operating systems to provide expanded support for Local Area Networking.

The design of Microsoft OS/2 recognizes this fact by providing substantial networking support in the form of the Microsoft OS/2 LAN Manager.

Basically, the MS OS/2 LAN Manager is a protected mode version of Microsoft *Networks*, and it is compatible with existing versions of MS-Net, XENIX Net and the IBM PC Local Area Network Program.

The LAN Manager is a separately "installable" component of MS OS/2; once installed, it works with the operating system to seamlessly solve networking requirements.

The LAN Manager is designed to facilitate the traditional network file- and printer-sharing services. But it goes far beyond this by providing for a new class of communication facilities, which will allow software developers to increase the sophistication of network services substantially and will make distributed processing practical.

The LAN Manager uses the protected mode, multitasking facilities of MS OS/2 to implement non-dedicated servers with extensive network administration and security services, as well as transparent file- and printer-sharing capabilities.

The LAN Manager allows an OS/2 based system to be added to existing MS-Net-based networks. The new system can function within the network as a workstation or a server or both.

Protected Mode NetBios

The multitasking environment in which the LAN Manager must function requires a new, protected-mode version of the MS-Net NetBios. The NetBios is the equivalent of the transport layer of the OSI model, and provides for virtual circuit and datagram services.

The standardized NetBios API allows network drivers to access low-level hardware (for such tasks as checking the status of a network card). The NetBios is designed not only to understand the hardware at one end (network hardware comes with a customized Hardware/NetBios layer), but also to understand API calls from network drivers at the other end. Applications access NetBios services via the network drivers.

The LAN Manager must deal with several multitasking issues. First, applications will most likely need to share access to more than one network driver. Second, there must be a way for network drivers to share access to the NetBios services.

Central to the LAN Manager is the Microsoft Net Kernel, which provides applications with the support necessary to simultaneously access one or more network drivers. Very briefly, the kernel serves as a dispatcher for applications' requests to access network drivers, and controls access to those network drivers by treating each driver as a named, installable device driver that can be opened or closed.

To insure that network drivers can interface to the net kernel and therefore share access to the NetBios, developers must adhere to the system interface specifications (and the protected mode NetBios API) published as part of the Microsoft OS/2 LAN Manager. These specifications will ease the effort that is necessary for OEMs and third-party network software developers to design their hardware and software applications to support the new multitasking, distributed processing environment.

Distributed Processing

From a software developer's point of view, the LAN Manager can be seen as providing a set of extensions to the MS OS/2 Inter-process Communications API. Just as MS OS/2 provides semaphores, pipes, and queues for communication between different processes running on the same machine, the LAN Manager provides named pipes and mailslots for communication between processes running on different machines on the network.

Named pipes and mailslots are the building blocks of distributed processing. These remote inter-process communication facilities provide the ability to establish communications channels between tasks,

Figure 1: DosMakeNmPipe Function

int FAR PASCAL

DosMakeNmPipe(name, handle, omode, pmode, size1, size2, timeout)

```
char far * name;          asciz pipe name
int far * handle;         place for handle to be returned
unsigned int omode;       DOS open mode
unsigned int pmode;       pipe open mode
unsigned int size1;       hint of outgoing buffer size
unsigned int size2;       hint of incoming buffer size
long     timeout;         time out for DosWaitNmPipe
```

Name: Asciz name of pipe. Pipes are named \PIPE\NAME.

Handle: Handle of the named pipe that is created.

Omode: Open mode mask as for DosOpen call. The following bits are defined:

Open Mode Bits

```
5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
O W * * * * * * I O O O * A A A
```

I Inheritance:
- 0 = Spawned processes inherit the pipe handle
- 1 = Spawned processes don't inherit the pipe

W Write-through:
- 0 = Write-behind to remote pipes is allowed
- 1 = Write-behind to remote pipes is not allowed

AAA Access mode:
- 000 = Inbound pipe (client to server)
- 001 = Outbound pipe (server to client)
- 010 = Full-duplex pipe (server to/from client)
- Other values invalid

Pmode: Pipe-specific mode parameters

```
5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
B * * * T T R R |... I count..|
```

B Blocking:
- 0 = Reads/writes block if no data available
- 1 = Reads/writes return immediately if no data

Reads normally block until at least partial data can be returned. Writes by default block until all bytes requested have been written.

Nonblocking mode (B=1) changes this behavior as follows:

1) Reads will return immediately with BytesRead = 0 if no data is available.

2) Writes will return immediately with BytesWritten = 0 if the data transfer cannot be started. Otherwise, the entire data area will be transferred.

TT Type of a pipe:
- 00 = Pipe is a byte stream pipe
- 01 = Pipe is a message stream pipe

All writes to message stream pipes record the length of the write along with the written data.

RR Read mode
- 00 = Read pipe as a byte stream pipe
- 01 = Read pipe as a message stream

Message pipes can be read as byte or message streams, depending on the setting of RR. Byte pipes can only be read as pipe streams

Icount: 8-bit count to control pipe instancing. When making the first instances of a named pipe, Icount specifies how many instances can be created: 1 means that this can be the only instance (pipe is unique) and -1 means the number of instances is unlimited; 0 is a reserved value. Subsequent attempts to make a pipe will fail if the maximum number of allowed instances already exists. The Icount parameter is ignored when making other than the first instance of a pipe. When multiple instances are allowed multiple clients can simultaneously DosOpen to the same pipe name and get handles to distinct pipe instances.

Size1: Hint to system, number of bytes to allocate for outgoing buffer.

Size2: Hint to system, number of bytes to allocate for incoming buffer.

Timeout: Default value for timeout parameter to PipeWait. This value may be set only at the creation time of the first instance of the pipe name. If at that time the value is zero, a systemwide default value (50 ms) will be chosen.

allowing for sharing of data—not only between a server machine and its clients, but also among peer machines residing on a given network.

Data-base applications provide a clear example of the distinction between file service and distributed proessing. Using current LAN technology, it is typical to search a file resident on another machine by reading a virtual file that is mapped to the real file by the MS-Net redirector.

In the distributed processing model, the requesting workstation passes a request to the server, which then conducts the search and passes back the results. The only information that travels on the network is the original request and the reply. This requires a level of inter-machine communication not possible with file- or record-oriented facilities.

The LAN Manager API is analogous at the network level to the MS OS/2 Inter-process Communication API that allows thread-to-thread and process-to-process communication at the operating system level. At any given moment, LAN API functions can block, be blocked, or can be in a state of execution (i.e., reading, writing or querying either named pipes or mailslots).

At the applications level, the LAN API provides CALL-based functions that can be incorporated into high-level language code. Programs written to the specifications of the API will be able to communicate across a network.

Virtual Circuits

The LAN API supports communications at the "Network and Trasnsport Levels" of the Open System Interconnect (OSI) model. Stream-oriented and optionally, message-oriented information is communicated through a mechanism called the named pipe. Message-oriented information can also be communicated via the high-speed mechanism called a named mailslot.

To understand the difference between named pipes and mailslots, one needs to understand two concepts basic to communications: those of virtual circuits and datagrams.

Virtual circuits are network-level data channels that guarantee an error-free and continuous communication channel, where each packet of routed information is delivered in order. Regardless of how a channel is established between two points (it could, for example, be a very complex switching network in a phone system), the two communicating ends perceive the channel as a direct communications link. This is, in essence, how named pipes work.

Datagrams are network-level channels that accept isolated messages (specific packets of information) from the transport layer and attempt to deliver them without regard for order or timeliness, much the same way as a postal system, in which the sender mails letters without knowing exactly when they will arrive, or in what order (message B, sent after message A, might arrive before message A). Named mailslots work in this fashion.

Across the network, either of these mechanisms lets processes transparently establish remote or local communications through symbolic names that follow MS OS/2 file naming conventions. Additionally, it is possible to impose access restrictions on named pipes and mailslots, just as can be done with files.

Named Pipes

Every reader of *MSJ* should be familiar with MS-DOS pipes. These pipes function as a transparent holding area for the output of one program to become the input of another. For example, DIR | SORT routes the output from DIR to a temp file that is then read by SORT.

Named pipes essentially extend this facility to provide direct communication between processes running on a network. Named pipes are two-way (or full duplex) channels that can be written at either end and read at the opposite end (that is, what is written at one end cannot be read from that same end).

To create a named pipe a process executes the DosMakeNmPipe function call (see **Figure 1**), which then opens a pipe as either a byte stream or a message stream. DosMakeNmPipe returns an identifying handle for that end of the pipe, which is called the serving end of the pipe. DosOpen (a regular OS/2 API call) opens the other end of the pipe, known as the client end, and similarly returns a handle.

Once a named pipe has been established it is controlled through the facilities provided by the LAN Manager. It can be read, written to and queried for activity or information. Named pipes can, at this point be utilized across a network, and

can be serially closed and opened by other processes.

Named pipes are therefore extremely flexible. Once created they can be accessed just like any sequential file, from beginning to end. Because rewinding of pipes is not possible (information comes and goes), the contents of a pipe can only be accessed once, unless the original information is again sent through the pipe.

Local and remote procedure call dialogues between processes can be efficiently implemeted because named pipes support transaction I/O calls.

Named pipes can also be accessed by applications running on MS DOS 3.*x*, machines serving as workstations on a given network, allowing those programs to communicate with server-based applications.

Thus, DosMakeNmPipe can allow multiple instances of a pipe, each with the same name. This insures that multiple clients DosOpening to that name will obtain separate and distinct pipes to the serving process.

Named Mailslots

Named pipes must insure that a full duplex communication channel exists between processes. There is, of necessity, a considerable amount of system overhead necessary to maintain such a connection.

Certain types of application-specific transactional messages to be exchanged between processes do not require a full duplex, error-free channel of communication. A named mailslot provides the optimal means for handling messages under these conditions. Note that byte streams cannot take advantage of named mailslots. Byte streams need the support mechanisms built into named pipes to ensure error-free transmission.

Mailslots are inherently fast because they need not be opened or closed by processes. Once a named mailslot has been established through the DosMakeMailslot function call processes running on different machines across a network simply write to it by name.

The function call used to write to a named mailslot, DosWriteMailslot, supports both "class" and "priority" settings for messages. Priority controls which messages DosReadMailslot reads first.

Class is usually used with remote mailslots. Messages can be assigned either a first- or second-class status. First-class mail is reliable in that DosWriteMailslot blocks until a message is delivered or an error occurs, insuring that the state of the message will be known. Second-class mail is sent strictly without error checking. Although delivery will probably succeed, if it fails an error will not be reported.

Between high-speed named mailslots and highly reliable named pipes, the LAN Manager provides the necessary communications facilities necessary to meet most sophisticated network communications needs. Software developers now have a base on which to develop complete, distributed applications. The opportunity to take advantage of a programming environment in which applications can access remote sites, and in which workstations can offload work to a server, will drive the next generation of software. **—T.R.**

A Compleat Guide to Writing Your First OS/2 Program

Charles Petzold

For the program developer, OS/2 opens up a whole new world of capabilities and challenges, and along with them, new concepts to learn. We'll explore some of these new concepts by walking through the creation of a typical OS/2 application that demonstrates inter-process communication (IPC) and multitasking.

FINDDEMO and FINDER work in conjunction to locate and list all files on a disk that fit a particular file specification, for instance, *.EXE. You can enter up to nine different file specifications at any one time, and all nine run simultaneously. **Figure 1** shows FINDDEMO in action.

FINDDEMO.EXE is the program that you execute from the OS/2 command line. FINDER.EXE must be in the current subdirectory or a subdirectory in your PATH. FINDDEMO manages the screen display and keyboard input and loads as many as nine instances of FINDER.EXE as child processes. FINDER.EXE does the actual file searching and funnels data back to FINDDEMO via a queue. Separate threads within FINDDEMO read the queue and monitor termination of the FINDER instances.

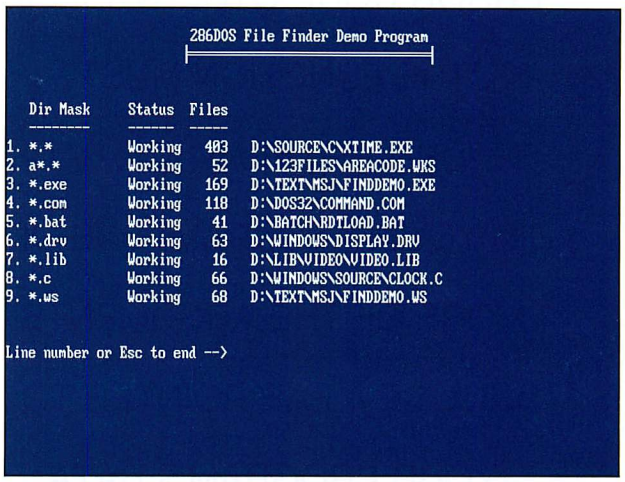
In other words, there's a real party going on here. I'll have more to say about the workings of this program after we look at what's involved in coding, compiling, and linking it. If you've already done some *Windows* programming, don't be surprised to see some familiar things along the way—you're several steps ahead of everybody else.

Protected Mode

Your first OS/2 programs will probably be existing MS-DOS programs that you will port to OS/2. Most programs written in straight C with standard library functions can simply be recompiled by using the new .LIB files.

However, programs with assembly language routines that make software interrupt calls, programs that use the Microsoft C int86 or intdos library functions, and programs that access memory outside the program or directly manipulate I/O ports will have to be modified somewhat.

OS/2 supports nearly 200 function calls that you'll use for communicating with the


286DOS File Finder Demo Program
Dir Mask Status Files
1. *.* Working 403 D:\SOURCE\C\XTIME.EXE
2. a*.* Working 52 D:\123FILES\AREACODE.WKS
3. *.exe Working 169 D:\TEXT\MSJ\FINDDEMO.EXE
4. *.com Working 118 D:\DOS32\COMMAND.COM
5. *.bat Working 41 D:\BATCH\RDTLOAD.BAT
6. *.drv Working 63 D:\WINDOWS\DISPLAY.DRV
7. *.lib Working 16 D:\LIB\VIDEO\VIDEO.LIB
8. *.c Working 66 D:\WINDOWS\SOURCE\CLOCK.C
9. *.ws Working 68 D:\TEXT\MSJ\FINDDEMO.WS
Line number or Esc to end -->

▲ **Figure 1** **FINDDEMO locates a file on a hard disk that fits a particular file specification. It can work with nine separate file specifications simultaneously.**

YOUR FIRST OPERATING SYSTEM/2 (OS/2) PROGRAMS WILL PROBABLY BE EXISTING MS-DOS PROGRAMS THAT YOU WILL PORT TO OS/2. MOST PROGRAMS WRITTEN IN STRAIGHT C WITH STANDARD LIBRARY FUNCTIONS CAN SIMPLY BE RECOMPILED BY USING THE NEW .LIB FILES.

Charles Petzold is a writer and columnist for several computer magazines.

VIRTUALLY ALL OS/2 FUNCTION CALLS RETURN A ZERO IF THE FUNCTION IS SUCCESSFUL AND AN ERROR CODE OTHERWISE. IF THE FUNCTION MUST RETURN INFORMATION BACK TO YOUR PROGRAM, ONE OR MORE OF THE ARGUMENTS WILL BE A FAR POINTER TO YOUR PROGRAM'S DATA SEGMENT. OS/2 USES THIS POINTER TO STORE THE RESULT OF THE FUNCTION. SOME OS/2 FUNCTION CALLS ALSO REQUIRE FAR POINTERS TO STRUCTURES FOR RETURNING MORE EXTENSIVE INFORMATION.

operating system and the PC's hardware. Obviously, these new function calls encompass virtually everything you can do now under MS-DOS 3.*x* using Int 21H. However, they also include a much faster and more versatile set of character-mode video I/O calls than the calls that have previously been available from the IBM PC BIOS services. You will no longer have to directly access video display memory in order to get good character-mode performance. OS/2 also includes a set of mouse function calls and many new calls that are related to tasking and inter-process communication.

DOS Function Calls

In C programs, OS/2 function calls look like ordinary function or subroutine calls. The names of all the functions are capitalized, and they begin with the letters DOS, KBD (keyboard), VIO (video), or MOU (mouse).

Virtually all OS/2 function calls return a zero if the function is successful and an error code otherwise. If the function must return information back to your program, one or more of the arguments will be a far pointer to your program's data segment. OS/2 uses this pointer to store the result of the function. Some OS/2 function calls also require far pointers to structures for returning more extensive information.

For instance, if you want to obtain the current cursor position, the OS/2 function that you must use is VIOGETCURPOS. The first argument is a far pointer to a variable where OS/2 stores the cursor row position, the second argument is a far pointer to a variable to store the column, and the third parameter is a "video I/O handle" that currently must be set to zero.

Within a C program you would define two variables for the row and column cursor positions like this:

```
unsigned row, col ;
```

and then call VIOGETCURPOS by using pointers to these variables:

```
VIOGETCURPOS
  (&row, &col, 0) ;
```

The C source code files for FINDDEMO, shown in **Figure 2**, and FINDER, shown in **Figure 3**, begin with the statement

```
#include <doscall.h>
```

This DOSCALL.H header file contains just two lines:

```
#include
   <doscalls.h>
#include
   <subcalls.h>
```

In the DOSCALLS.H and SUBCALLS.H header files are the declarations for all OS/2 functions. The DOS functions are declared in DOSCALLS.H, and the KBD, VIO, and MOU functions are declared in SUBCALLS.H. For instance, the VIOGETCURPOS function is declared in SUBCALLS.H as

```
extern unsigned
far pascal
VIOGETCURPOS
  (unsigned far *,
  unsigned far *,
  unsigned) ;
```

All OS/2 functions are declared far, which directs the compiler to generate a far intersegment call, and pascal. The pascal keyword indicates a Pascal calling sequence—the arguments are pushed onto the stack from left to right, and the called function then adjusts the stack before returning control to the caller.

Normally, C compilers generate code that pushes arguments on the stack from right to left, and the caller fixes the stack. This calling sequence permits C to support functions with a variable number of arguments. Moreover, you can usually call a C function with an incorrect number of parameters, and while the function may not work correctly, in many cases the program will not crash.

A function that uses the Pascal calling sequence, however, expects to receive a fixed number of arguments. Therefore, if you get the compiler warning message "too few actual parameters" for an OS/2 function call, you should fix the code before you try to run the program.

Note that the function declaration shows the first two parameters to be far pointers to unsigned integers, but the VIOGETCURPOS call used simply &row and &col. The C compiler generates far addresses without casting based on the function declaration.

The DOSCALLS.H and SUBCALLS.H header files also contain structure declarations used in some of the other OS/2 function calls. For instance, FINDER.EXE uses the FileFindBuf structure in conjunction with the DOSFINDFIRST and DOSFINDNEXT function calls. FINDDEMO.C uses the ResultCodes structure for returning values from the DOSEXECPGM and DOSCWAIT functions, and the KeyData structure for returning keyboard input from KBDCHARIN. Here's what the KeyData structure declaration looks like:

```
struct KeyData
  {unsigned char
    char_code ;
```

```
/* finddemo.c -- program to demonstrate IPC and multitasking */

#include <doscall.h>

#define WORD  unsigned int
#define DWORD unsigned long

#define NUMPROC     9                   /* number of process */
#define CHILDPROG   "FINDER.EXE"
#define QUEUENAME   "\\QUEUES\\FINDDEMO"
#define THREADSTACK 1024

void far dispthread (void) ;
void far waitthread (void) ;

WORD   queuehandle, count [NUMPROC] ;
DWORD  runsem [NUMPROC], waitsem [NUMPROC] ;
struct ResultCodes rc [NUMPROC] ;
struct {
     WORD count ;
     struct {
          WORD  reserved ;
          DWORD far *sem ;
          } index [NUMPROC] ;
     } semtab ;

main ()
     {
     static char prompt[ ] = "Line number or Esc to end -->  \b" ;
     struct KeyData keydata ;
     WORD   index, len, i, dispID, waitID, sigaction ;
     DWORD  sigaddr ;
     char   dirmask [15], dispstack [THREADSTACK],
            waitstack [THREADSTACK] ;

     /*
         Initialize: Set up "semtab" structure for DOSMUXSEMWAIT.
         ----------  Disable Ctrl-Break and Ctrl-C exits.
                     Create queue for IPC with FINDER.EXE.
                     Create threads for messages from FINDER
                         and waiting for FINDER terminations.
                     Display text.
     */

     semtab.count = NUMPROC ;
     for (index = 0 ; index < NUMPROC ; index++) {
          DOSSEMSET ((DWORD) &waitsem[index]) ;
          semtab.index[index].sem = &waitsem[index] ;
          }
     DOSSETSIGHANDLER (0L, &sigaddr, &sigaction, 1, 1) ;
     DOSSETSIGHANDLER (0L, &sigaddr, &sigaction, 1, 4) ;

     if (DOSCREATEQUEUE (&queuehandle, 0, QUEUENAME)) {
          puts ("FINDDEMO: Cannot create new queue") ;
          DOSEXIT (1, 1) ;
          }
     if (DOSCREATETHREAD (dispthread, &dispID,
                          dispstack + THREADSTACK) ||
         DOSCREATETHREAD (waitthread, &waitID,
                          waitstack + THREADSTACK))
          {
          puts ("FINDDEMO: Cannot create threads") ;
          DOSEXIT (1, 1) ;
          }
     displayheadings () ;

               /*
                    Main Loop: Display prompt and read keyboard.
                    ---------  Execute FINDER.EXE.
               */
     do   {
```

CONTINUED

▲ **Figure 2** FINDDEMO.C is the source code for the main FINDDEMO program.

Figure 2 CONTINUED

```
          VIOSETCURPOS (18, 0, 0) ;
          VIOWRTTTY (prompt, sizeof prompt - 1, 0) ;
          KBDCHARIN (&keydata, 0, 0) ;

          index = keydata.char_code - '1' ;

          if (index <= NUMPROC && rc[index].TermCode_PID == 0) {
               VIOWRTTTY (&keydata.char_code, 1, 0) ;
               VIOWRTNCHAR (" ", 77, 7 + index, 3, 0) ;
               do   {
                    VIOSETCURPOS (7 + index, 3, 0) ;
                    len = 13 ;
                    KBDSTRINGIN (dirmask, &len, 0, 0) ;
                    }
               while (len == 0) ;

               dirmask [len] = '\0' ;
               executeprogram (index, dirmask) ;
               }
          }
     while (keydata.char_code != 27) ;

               /*
                    Clean-up: Kill all existing FINDER.EXE processes.
                    --------  Wait for processes to terminate.
                              Close the queue and exit.
               */

     for (index = 0 ; index < NUMPROC ; index++)
          if (rc[index].TermCode_PID)
               DOSKILLPROCESS (0, rc[index].TermCode_PID) ;

     for (index = 0 ; index < NUMPROC ; index++)
          DOSSEMWAIT ((DWORD) &runsem [index], -1L) ;

     DOSCLOSEQUEUE (queuehandle) ;
     DOSEXIT (1, 0) ;
     }

displayheadings ()
     {
     static char heading  [ ] = "286DOS File Finder Demo Program",
                 colheads [ ] = "Dir Mask     Status  Files",
                 colunder [ ] = "--------     -----  ----" ;
     char        buffer  [5] ;
     WORD        row, col, i, len ;

     VIOGETCURPOS (&row, &col, 0) ;            /* get current attr */
     VIOWRTTTY (" ", 1, 0) ;
     len = 2 ;
     VIOREADCELLSTR (buffer, &len, row, col, 0) ;
     VIOSCROLLUP (0, 0, -1, -1, -1, buffer, 0) ;   /* clear screen */

     len = sizeof heading - 1 ;
     col = (80 - len) / 2 ;
     VIOWRTCHARSTR (heading, len, 1, col, 0) ;          /* heading */
     VIOWRTNCHAR   ("\xC6",   1, 2, col - 1,   0) ;   /* underline */
     VIOWRTNCHAR   ("\xCD", len, 2, col,       0) ;
     VIOWRTNCHAR   ("\xB5",   1, 2, col + len, 0) ;
     VIOWRTCHARSTR (colheads, sizeof colheads - 1, 5, 3, 0) ;
     VIOWRTCHARSTR (colunder, sizeof colunder - 1, 6, 3, 0) ;

     for (i = 0 ; i < NUMPROC ; i++) {                  /* numbers */
          sprintf (buffer, "%d.", i + 1) ;
          VIOWRTCHARSTR (buffer, 2, 7 + i, 0, 0) ;
          }
     }

executeprogram (index, dirmask)
     WORD index ;
     char *dirmask ;
     {
     char objbuf [32] ;
     char args [128] ;
```

CONTINUED

```
  unsigned char
    scan_code ;
  unsigned char
    status ;
  unsigned char
    nls_shift ;
  unsigned
    shift_state ;
  unsigned long
    time} ;
```

That's a little more information than you currently get from Int 16H, isn't it?

Assembly Language

If you are programming in assembly language, OS/2 function calls are made by pushing arguments on the stack and calling external far functions. Near the top of your source code you specify that the OS/2 function is in a different segment:

```
extrn
  VIOGETCURPOS:far
```

The two variables to receive the row and column positions would go in your data segment:

```
row  dw   ?
col  dw   ?
```

When you call VIOGETCURPOS, you push the arguments on the stack and make the call:

```
push ds
push offset row
push ds
push offset col
push 0
call VIOGETCURPOS
```

When you return from the VIOGETCURPOS call, the AX register contains the return value of the function call, and the stack pointer is the value prior to pushing the arguments on the stack.

This code sample pushes immediate values on the stack. Because these instructions are supported by the 80286 but not by the 8086, you have to include the assembler directive

```
.286c
```

in your source code file. Or you can simply transfer the values to registers before pushing them on the stack.

Linking

The new LINK generates a "new-format executable." The extension is still .EXE, but the format of the file is not the same as that used in MS-DOS versions through 3.2. However, it *is* the same as the new-format executable currently in use for *Windows* applications.

Among other things, the new-format executable contains an "import table" that OS/2 uses to match the far calls in your program code to the appropriate OS/2 function calls. The actual entry address of the function call routine is inserted into the code when the program is loaded into memory. (This is dynamic linking, which has appeared previously in *Windows* programming.)

LINK requires that two libraries be explicitly listed in the command line. These are SLIBC5.LIB, which contains normal C library functions that make OS/2 functions calls, and DOSCALLS.LIB.

The DOSCALLS.LIB library is an "import library." It merely provides LINK with the module names and ordinal numbers associated with the OS/2 function calls. LINK uses this information to set up the import table contained in the new-format executable. DOSCALLS.LIB performs the same function in OS/2 programs that the SLIBW.LIB library performs in *Windows* applications.

Module Definition File

The new LINK also requires that a module definition file be specified as the fifth argument on the LINK command line. The module definition file (yet another concept that *Windows* programmers are already familiar with) provides information about your program's segments, stack size, and local heap size. The two module definition files that describe FINDDEMO.DEF and FIND-

Figure 2 CONTINUED

```
strcat (strcpy (args, CHILDPROG), " ") ;     /* construct args */
     strcat (strcat (args, dirmask), " ") ;
     strcat (strcat (args, QUEUENAME), " ") ;
     itoa (index, args + strlen (args), 10) ;

     count [index] = 0 ;                          /* initialize count */

     if (DOSEXECPGM (objbuf, 32, 2, args, 0, &rc[index], CHILDPROG))
          {
          puts ("FINDDEMO: Can't run FINDER.EXE") ;
          DOSEXIT (1, 1) ;
          }
     VIOWRTCHARSTR ("Running", 7, index + 7, 16, 0) ;/* executing */
     DOSSEMSET    ((DWORD) &runsem [index]) ;
     DOSSEMCLEAR ((DWORD) &waitsem[index]) ;
     }

void far dispthread ( )     /* thread to read messages from FINDER */
     {                      /*   and display filenames.            */
     DWORD request ;
     WORD  len, index, i ;
     char  far *farptr ;
     char  priority, pathname [80], buffer [64] ;

     while (1) {
          DOSREADQUEUE (queuehandle, &request, &len,
                        &(DWORD)farptr, 0, 0, &priority, 0L) ;
          i = 0 ;
          while (pathname [i++] = *farptr++) ;
          index = (WORD) (request >> 16) ;
          count [index] += len > 0 ;
          sprintf (buffer, "%5d   %-48.48s",
                  count [index], pathname) ;
          VIOWRTCHARSTR (buffer, 56, 7 + index, 24, 0) ;
          DOSFREESEG ((WORD) ((DWORD) farptr >> 16)) ;
          }
     }

void far waitthread ( )  /* thread to wait for FINDER terminations */
     {
     WORD   index, PID ;
     struct ResultCodes rescode ;

     while (1) {
          DOSMUXSEMWAIT (&index, (WORD far *) &semtab, -1L) ;
          DOSCWAIT (0, 0, &rescode, &PID, 0) ;

          for (index = 0 ; index < NUMPROC ; index++) /* find index */
               if (PID == rc[index].TermCode_PID)
                    break ;

          VIOWRTCHARSTR (rescode.TermCode_PID ? "Halted " : "Done   ",
                              7, index + 7, 16, 0) ;
          rc[index].TermCode_PID = 0 ;
          DOSSEMCLEAR ((DWORD) &runsem [index]) ;
          DOSSEMSET    ((DWORD) &waitsem[index]) ;
          }
     }
```

```
/* finder.c -- child process of finddemo */

#include <doscall.h>

unsigned index ;
unsigned queuehandle ;
unsigned parentPID ;

main (argc, argv)
     int  argc ;
     char *argv [] ;
     {
     if (argc < 4) {
          puts ("Run FINDDEMO rather than FINDER") ;
          DOSEXIT (1, 1) ;
          }
     if (DOSOPENQUEUE (&parentPID, &queuehandle, argv [2])) {
          puts ("FINDER: Cannot open queue") ;
          DOSEXIT (1, 1) ;
          }
     index = atoi (argv [3]) ;

     writequeue ("") ;
     chdir ("\\") ;
     find (argv [1]) ;
     writequeue ("") ;

     DOSCLOSEQUEUE (queuehandle) ;
     DOSEXIT (1, 0) ;
     }

find (searchstr)
     char      *searchstr ;
     {
     struct    FileFindBuf ffb ;
     char      cwd [81], pathname [100] ;
     unsigned handle = 0xFFFF, num = 1 ;

     if (cwd [strlen (getcwd (cwd, 80)) - 1] != '\\')
          strcat (cwd, "\\") ;

     DOSFINDFIRST (searchstr, &handle, 7, &ffb, sizeof ffb, &num, 0L) ;
     while (num) {
          writequeue (strcat (strcpy (pathname, cwd), ffb.file_name)) ;
          DOSFINDNEXT (handle, &ffb, sizeof ffb, &num) ;
          }
     DOSFINDCLOSE (handle) ;

     handle = 0xFFFF ;
     num  = 1 ;
     DOSFINDFIRST ("*.*", &handle, 0x17, &ffb, sizeof ffb, &num, 0L) ;
     while (num) {
          if (ffb.attributes & 0x10 && ffb.file_name [0] != '.') {
               chdir (ffb.file_name) ;
               find (searchstr) ;
               chdir ("..") ;
               }
          DOSFINDNEXT (handle, &ffb, sizeof ffb, &num) ;
          }
     DOSFINDCLOSE (handle) ;
     }

writequeue (str)
     char *str ;
     {
     unsigned selector, parentselector ;
     char far *farptr ;
     int      len = strlen (str) ;

     DOSALLOCSEG (len + 1, &selector, 1) ;
     farptr = (char far *) (((unsigned long) selector) << 16) ;

     while (*farptr++ = *str++) ;

     DOSGIVESEG (selector, parentPID, &parentselector) ;
     DOSFREESEG (selector) ;
     farptr = (char far *) (((unsigned long) parentselector) << 16) ;

     DOSWRITEQUEUE (queuehandle, index, len, farptr, 0) ;
     }
```

▲ **Figure 3** FINDER.C is the source code for the child process executed by FINDDEMO.

ER.DEF are shown in **Figures 4** and **5**, respectively.

The NAME line indicates the name of the module, which is usually the same as that of the program. The DESCRIPTION is generally a copyright notice imbedded in the .EXE file. The keyword PROTMODE indicates that the program runs only in protected mode.

The CODE and DATA lines specify characteristics of the code and data segments in your program. The options that are shown here are normal for protected mode.

Both the code and data segments are MOVABLE. This is no sweat in protected mode because a segment can be physically moved but still retain the same selector address, previously known as the segment address. Specifying the CODE segment as PURE means that the same code segment loaded in memory can be used for multiple instances of the program. The automatic data segment of a program cannot be PURE because it contains the stack and most likely contains read/write data as well. However, data segments with read-only data without relocation information can also be flagged as PURE and can be shared among multiple instances of the program.

For large applications, you might divide the program into many different segments and specify characteristics of each in the module definition file. These characteristics would include the keywords PRELOAD, which means the segment is loaded into memory when the program is run, and LOADONCALL, which means the segment is loaded into memory only when a routine within it is needed. Based on a

least-recently-used algorithm, OS/2 can free up memory that is occupied by code segments and later reload them when needed from the .EXE file. This facility is essentially a built-in, hassle-free overlay manager.

The automatic data segment of your program contains static variables, the stack, and a local heap organized as in **Figure 6**.

The stack is fixed to the size specified in the STACKSIZE line of the module definition file. The HEAPSIZE value is a minimum local heap size, which is the size that OS/2 sets aside when your program is first loaded into memory. If you use C memory allocation functions such as malloc to allocate more memory from your local heap than is specified by HEAPSIZE, OS/2 can expand the local heap—by physically moving the data segment in memory if necessary.

MAKEing the Program

A make-file for the FINDDEMO application is shown in **Figure 7**. You create the FINDDEMO.EXE and FINDER.EXE executables by running

```
MAKE FINDDEMO
```

The compile step uses several compiler switches:

```
cl -c -G2 -Zp
  finddemo.c
```

The -c switch specifies that the program is to be compiled but not linked; this is necessary because the link step requires nondefault arguments. The -G2 switch compiles code by using 80286 instructions. This is optional, but it results in a slightly smaller .EXE size and faster execution.

The -Zp switch indicates that structures are "packed." Normally the C compiler aligns each element of a structure on an even address. However, some of the structures declared in DOSCALLS.H and SUBCALLS.H, such as the KeyData structure shown earlier, contain consecutive char variables.

```
NAME            FINDDEMO
DESCRIPTION     'File Finder Demonstration Program'
PROTMODE
DATA            MOVABLE
CODE            MOVABLE PURE
HEAPSIZE        2048
STACKSIZE       4096
```

▲ **Figure 4** **FINDDEMO.DEF is the module definition file for the FINDDEMO program.**

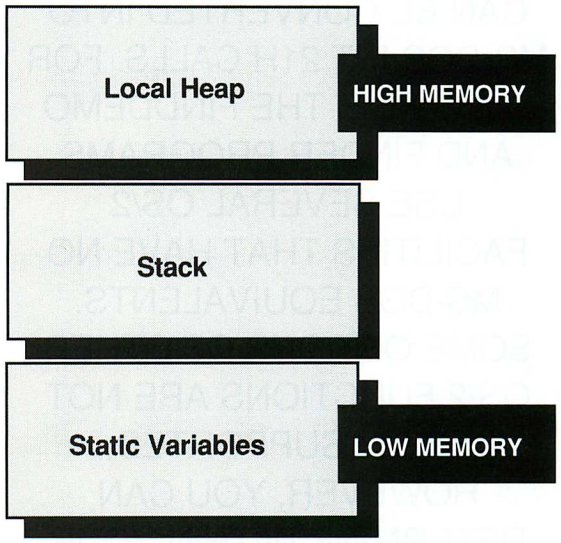
```
NAME            FINDER
DESCRIPTION     'File Finder Module for FINDDEMO'
PROTMODE
DATA            MOVABLE
CODE            MOVABLE PURE
HEAPSIZE        2048
STACKSIZE       8192
```

▲ **Figure 5** **FINDER.DEF is the module definition file for the FINDER program.**

New Executable

The new-format executable created by LINK is really a superset of the old-format executable. It begins with an old-format header. This makes it appear to old DOS versions (and the OS/2 compatibility box) as a normal .EXE program. However, when you link the FINDDEMO and FINDER programs, the old-format header is constructed so that the program appears to require more memory than is available. So, if you attempt to run FINDDEMO.EXE under real mode (the OS/2 compatibility box or MS-DOS 2.*x* or 3.*x*), you get the message "Program too big to fit in memory."

You have some alternatives to this message. The first al-



▲ **Figure 6** **Automatic data segment containing static variables, the stack, and a local heap.**

```
finddemo.obj : finddemo.
    cl -c -G2 -Zp finddemo.c

finddemo.exe : finddemo.obj finddemo.def
    link finddemo, /align:16, /map, doscalls slibc5, finddemo.def

finder.obj : findef.c
    cl -c -G2 -Zp finder.c

finder.exe : finder.obj finder.def
    link finder, /align:16, /map, doscalls slibc5, finder.def
```

▲ **Figure 7** This is the make-file used for creating the FINDDEMO.EXE and FINDER.EXE executable files.

NOT ALL OS/2 FUNCTIONS CAN BE CONVERTED INTO MS-DOS INT 21H CALLS. FOR INSTANCE, THE FINDDEMO AND FINDER PROGRAMS USE SEVERAL OS/2 FACILITIES THAT HAVE NO MS-DOS EQUIVALENTS. SOME OPTIONS OF OTHER OS/2 FUNCTIONS ARE NOT FULLY SUPPORTED. HOWEVER, YOU CAN DETERMINE WITHIN YOUR PROGRAM WHETHER IT IS RUNNING IN REAL MODE OR PROTECTED MODE AND HAVE SEPARATE LOGIC THAT IS APPROPRIATE FOR EACH.

ternative involves the STUB option in the module definition file, which *Windows* programmers already know about:

```
STUB 'oldprog.EXE'
```

The file oldprog.EXE is an old-format executable that runs under MS-DOS 2.*x* or 3.*x*. The STUB option directs LINK to insert this old-format executable in the top of the new-format executable. Now the .EXE file appears to MS-DOS 2.*x* or 3.*x* as a regular program. When MS-DOS 2.*x* or 3.*x* loads the program, this old program is executed instead.

Windows programmers generally include the line

```
STUB 'WINSTUB.EXE'
```

in their module definition files. The standard WINSTUB.EXE program simply displays the message "This program requires Microsoft Windows." You might want to create a similar program that would display a message such as "This program must be run in OS/2 protected mode."

The second alternative is to write a real-mode version and include that in the STUB line if your program is short. If you're writing a program using straight C without any explicit OS/2 function calls, you could simply compile and link it under a different name by using the old real-mode libraries. This process creates one .EXE file that actually contains two versions of the same program: the protected-mode version runs under protected mode, and the real-mode version runs under real mode.

For longer programs, or programs that use explicit OS/2 function calls, you should consider the third alternative: going family style.

Family Style

The OS/2 developer's kit includes a library file called API.LIB that translates many OS/2 function calls into equivalent MS-DOS Int 21H function calls. Video I/O calls are translated into equivalent BIOS calls and direct screen accesses. When you run

```
BIND program
```

after linking, parts of this API.LIB library are tacked onto the end of the .EXE file, and the old-format header is modified to run a loader program that is also inserted into the .EXE file. When you run the resulting .EXE file under a real-mode DOS version, the loader patches addresses into your program that call the routines within this library rather than OS/2 functions. If you run the program under OS/2 protected mode, the loader and library routines are simply discarded.

Thus, you have one executable file and one program that runs under protected-mode and real-mode MS-DOS. This is called the "family API" model.

Not all OS/2 functions can be converted into old MS-DOS Int 21H calls. For instance, the FINDDEMO and FINDER programs shown here use several OS/2 facilities that have no old MS-DOS equivalents. Some options of other OS/2 func-

tions are not fully supported. However, you can determine within your program whether it is running in real mode or protected mode and have separate logic that is appropriate for each.

The Workings

Now that we know how to create FINDDEMO.EXE and FINDER.EXE, let's look at how they work. FINDER is a child process run by FINDDEMO. FINDER does all the work while FINDDEMO sits back and waits for the messages from FINDER that report what it has found.

FINDDEMO starts off by creating a queue named \QUEUES\FINDDEMO. A queue is a linked list maintained by OS/2 that you can use to transfer data from one program to another. One program creates the queue; another program opens the same queue. The program that creates the queue—in this case, FINDDEMO—has read and write privileges. The program that then opens the existing queue—in this case, FINDER—has write privileges. Queues may be first-in-first-out (FIFO), last-in-first-out (LIFO), or based on priority; the FINDDEMO queue is FIFO.

FINDDEMO also creates two additional threads of execution. These appear in the program as functions called dispthread and waitthread. These two threads run simultaneously with the main thread, sometimes called the "parent" thread. I'll explain what they do a little later.

FINDDEMO then prompts for a number and a file specification from the user. For each file specification you enter, FINDDEMO executes FINDER by using the OS/2 function call DOSEXECPGM. FINDER is executed in an asynchronous mode—the DOSEXECPGM call returns immediately, and FINDDEMO can prompt for the next file specification. As many as nine instances of FINDER can be executed. Although these multiple instances use different data segments, they share the same code segment in memory.

When FINDDEMO executes FINDER it passes to it the file specification, the name of the queue, and an index number from 1 to 9 that indicates the display line on which the file specification appears.

Now let's look at FINDER. FINDER opens the queue and searches for files meeting the file specification over the entire disk by using a recursive function called find. You'll note that FINDER changes the subdirectory during this search. Under OS/2, each process maintains its own current subdirectory. So each instance of FINDER can change the subdirectory and not affect other instances. Nice, huh?

When FINDER finds a file that fits the file specification, it allocates a small shared segment of memory using DOSALLOCSEG and copies the pathname into it. The DOSGIVESEG function obtains a new selector (segment address) appropriate for its parent FINDDEMO. (FINDER obtained the process ID of its parent when it opened the queue.) FINDER can then write this segment to the queue.

Now back to FINDDEMO. During initialization, FINDDEMO created a second execution thread, the subroutine called dispthread. This thread sits in an infinite loop that starts off with a call to DOSREADQUEUE. When DOSREADQUEUE returns control to the program, it reads the contents of the shared memory segment created by FINDER and writes it to the display.

The other thread created during FINDDEMO initialization is waitthread. This is also an infinite loop that waits, using DOSCWAIT, for instances of FINDER to terminate. Because DOSCWAIT returns with an error if no child processes are running, FINDDEMO first sets nine semaphores and clears one only when executing FINDER. The waitthread thread is suspended, through the use of the function DOSMUXSEMWAIT, until one of these semaphores is cleared.

Another set of semaphores, which are in the array runsem, are normally cleared. They are set when an instance of FINDER is executed and cleared when the instance terminates. These semaphores are used when you terminate FINDDEMO by using the Esc key. FINDDEMO uses DOSKILLPROCESS to terminate any instances of FINDER that are still running and then waits for all of the runsem semaphores to be cleared.

The New Generation

I claimed earlier that FINDDEMO's use of inter-process communication and multitasking represented a "typical" OS/2 application. Obviously, you can take advantage of the larger protected mode addressing space and more versatile OS/2 API without getting into the fancy stuff. Whether you do get fancy or not, I'm sure you'll find some interesting applications for these new OS/2 capabilities. ❑

Microsoft OS/2 Software Development Kit

Software professionals will soon be able to explore the new operating system environment created by OS/2 when Microsoft presents the Microsoft Operating System/2 Software Development Kit, or SDK. The kit will include

- Microsoft OS/2 Version 1.0, which includes the Windows presentation manager
- The OS/2 LAN Manager,Version 1.0
- Microsoft C Optimizing Compiler, Version 4.5
- Microsoft Macro Assembler, Version 4.5
- The CodeView debugger, utilities, LINK, MAKE, LIB, IMPLIB, and a protected-mode text editor
- A one year subscription to Microsoft DIAL
- Attendance at one of the Microsoft OS/2 Developers Conferences
- Full documentation for all components of the kit

Since run time versions of the Windows presentation manager and the LAN manager are not yet ready to be released, the SDK only includes them as reference materials. Microsoft will offer updates to the run time versions and all other components of the kit free of charge until the final OS/2 product is ready for distribution.

The DIAL system has been created for the purpose of giving developers fast and direct access to Microsoft both to report problems and to receive solutions (See MSJ Volume 2, No. 1 for a detailed description of DIAL).

The Microsoft C Optimizing Compiler Version 4.5 offers substantial new features that will allow applications to take full advantage of MS OS/2 while retaining full compatibility with Version 4.0. The major programming tools include new versions of the Microsoft C Compiler and Macro Assembler.

Fully compatible with the Microsoft Macro Assembler Version 4.0, MASM 4.5 is up to 25% faster in assembly speed. The assembler works under both OS/2 and MS-DOS, provides support for the CodeView debugger, simplified segmentation, and a new Mixed-Language Programming Guide and template programs for easy interface to C and other languages. MASM 4.5 supports the 386 instruction set, both for protected-mode and real-mode applications.

The Microsoft C Optimizing Compiler Version 4.5 offers new features that will allow applications to take full advantage of MS OS/2's protected, real, and compatibility modes while retaining full compatibility with Version 4.0. The new compiler contains a loop optimizer, which utilizes automatic enregistering, loop variants, and constant folding to increase the execution speed of programs by 20% to 30% over Version 4.0.

An integral part of the SDK will be free admission to a software developers conference. The seminar will give developers an opportunity to spend a considerable amount of time with Microsoft experts who can answer questions directly and in detail. Informal discussion groups are expected to be a part of the overall conference structure.

The primary purpose of the SDK is to make available to developers the tools necessary to begin working with MS OS/2. Microsoft Product Manager Mark Mackaman feels that "First, developers will port their existing MS-DOS applications to the new environment. Second, after the first rush of applications porting you will see the next generation of products for MS OS/2, which will offer dramatic improvements in functionality and ability, similar to the leap from VisiCalc to Lotus 1-2-3."

The software provided by the SDK will run on the IBM PC AT, the Compaq DeskPro 286 and 386 machines, and most other true compatibles. The software requires a minimum of 1.5 Mb of RAM, a high density (96 TPI) disk drive, and a hard disk. A graphics display and adapter are also recommended.

The cost of the complete SDK package is $3,000. To be made available in early summer, the SDK will provide all of the tools necessary to begin preparing for the leap to the next generation.

For more information, including information on ordering, you can contact Microsoft directly: Microsoft Corporation, Dept. MS OS/2 SDK, 16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717.

An Interview with Gordon Letwin

OS/2: Turning Off the Car to Change Gears

Lori Valigra

The development of Microsoft's multitasking operating system has taken many twists and turns since Microsoft began working on it three years ago. The many stages it has gone through so far have resulted from working with the schizophrenic Intel 80286 chip, which can run in either of two incompatible modes: protected (286) mode, and unprotected (8086 or "real") mode. The main problem has been to get an operating system which runs both protected- and real-mode programs. Microsoft solved the problem with mode switching—a feature which essentially works by mimicking the act of turning off the computer and restarting it, similar to turning off a car in order to switch gears.

The new operating system began as an ambitious project to produce an operating system for Intel 80286-based computers running a mixture of office automation applications. This new operating system was to be capable of multitasking and compatible with the protected 286 mode. But as the project got too unwieldly, Microsoft split it up in early 1985, announcing a version called DOS 4. This version incorporates multitasking but does not include protected mode features. The DOS 5 project, as it was called, is intended to support the protected mode and add other features.

OS/2, as the project has finally been named, was easily the most difficult and challenging program written at Microsoft to date, with more than 350,000 lines of code. While Microsoft generally feels that the most efficient projects are those staffed by one person, the complexity and time constraints of the OS/2 project required that the company assign more than 35 programmers to it. Managing the project—dividing the programmers into teams, dividing the project among them, and coordinating the various teams' modifications to the program's code—became an enormous challenge. Adding to the challenge was the joint development of the product with IBM, a company whose culture and methodology are quite different from Microsoft's.

Microsoft Systems Journal talked with the project's chief architect, Gordon Letwin, to learn more about the planning,

OS/2 WAS EASILY THE MOST CHALLENGING PROGRAM WRITTEN AT MICROSOFT TO DATE, WITH MORE THAN 350,000 LINES OF CODE. WHILE MICROSOFT GENERALLY FEELS THAT THE MOST EFFICIENT PROJECTS ARE THOSE STAFFED BY ONE PERSON, THE COMPLEXITY AND TIME CONSTRAINTS REQUIRED THAT THEY ASSIGN MORE THAN 35 PROGRAMMERS TO OS/2.

Lori Valigra is a New York–based freelance writer, specializing in high technology, business, and international reporting.

THE INTEL 80286 IS DESIGNED TO RUN IN ONLY ONE MODE, EITHER REAL MODE OR PROTECTED MODE. SINCE WE KNEW WE NEEDED TO SUPPORT PROGRAMS WRITTEN USING PROTECTED MODE, A GROUP OF US, INCLUDING BILL GATES AND SOME PROJECT ENGINEERS, BRAINSTORMED TO TRY TO FIGURE OUT HOW TO RUN 8086 MODE APPLICATONS FROM WITHIN PROTECTED 286 MODE.

development, and cooperative effort that went into the development of OS/2.

MSJ: What was the goal of the OS/2 project?

Letwin: We tried to create the underpinnings for the ideal office automation operating system. In our opinion, that operating system would allow applications direct access to high-bandwidth peripherals, offer device-independent graphics drivers, provide multitasking without performance degradation (compared with a single-tasking system), run programs in both the protected 286 and real 8086 modes of the 286 processor, and provide a protected environment to assure system stability. I think we've done a pretty good job.

MSJ: Microsoft has been developing multitasking operating systems for three years now, beginning with DOS 4. What was the purpose of that operating system, and where does it stand now that OS/2 is out?

Letwin: DOS 4 was the first product to result from Microsoft's multitasking DOS effort. We began it even before IBM introduced the PC AT. It was an ambitious product that was originally to include a protected mode with mode switching abilities so it could run on the 8086 or the 286.

A general-purpose multitasking system needs to run in both modes: the unprotected 8086 mode so that we can run existing DOS applications, and the protected 286 mode so that we can multitask arbitrary, unrelated applications. But the architecture of the 286 caused some delays. Although we knew the project would be difficult, it was only after we'd gotten deeply into it that we realized just how difficult it would be.

As a result, DOS 4 became too complicated for our schedules. Because of the pressure of customer demand as well as that of previous commitments, we broke the project into two parts. DOS 4 runs only in real mode and provides multitasking only for specialized applications. DOS 5, which has now been released as OS/2, includes the protected mode and other features. DOS 4 was delivered in the last half of 1986 and is being sold in special application environments, primarily in Europe. It is a specialized product that can share the market with OS/2, because it runs on 8086 hardware, while OS/2 requires a 286. The move from DOS 4 to OS/2 was a gradual evolutionary process. OS/2 is by far the most complex and sophisticated operating system project we've worked on. To offer multitasking in a protected environment, we developed it jointly with IBM to get around the constraints of the 286.

MSJ: How does OS/2 compare with earlier multitasking operating systems like Concurrent CP/M?

Letwin: Unlike OS/2, Concurrent CP/M has a major problem. It serializes access to the file system. When multitasking you need to make the operating system features such as the file system accessible to multiple programs at a time. But Concurrent CP/M didn't do this. It only allowed one program to call the file system at a time. So in a real life situation with lots of I/O, Concurrent CP/M was too slow.

MSJ: What are some of the features of OS/2 that you consider particularly neat?

Letwin: One is the seamless system service interface, which means that system services don't need to be in the kernel. Some can be provided by subroutine libraries and others by system processes. Many services can be upgraded piecemeal, in the field, without changing the whole operating system. This is accomplished by a technique called "dynamic linking," which allows external procedure calls to be linked up at execution time, rather than link time. This same mechanism is used as the standard system interface between applications, utility packages, etc.

Another exceptional feature is direct hardware access, which allows an application to write directly to the hardware. For example, it will do special graphics on a screen.

MSJ: How did you get around the limitations of the 286 chip?

Letwin: The 286 is designed to run in only one mode, either real mode or protected mode. Since we knew we needed to support programs written using protected mode, a group of us, including Bill Gates and some project engineers, brainstormed to try to figure out how to run 8086 mode applications from within protected 286 mode. One option was "real mode emulation," where we could emulate the operation of 8086 real mode by doing a lot of special things to the protected mode segment tables. IBM was examining this technique independently. We thought emulation would be too slow, since the 286 would run at one-third the speed of an 8086. IBM's research later showed this to be the case. Soon thereafter, Intel began shipping a chip with a bug in it that would

GORDON LETWIN: THE CHALLENGE OF THE 286 SPEAKS HIS LANGUAGE

What attracted 34-year-old Gordon Letwin to computers in the late 1960s has also kept him in the forefront of programming today: an insatiable drive to overcome technical obstacles.

"I'd read about computers in the late 1960s, that they were supposed to be super complicated. Computers held an abstract mystique. And I wanted to understand how that stuff worked," he says. That was a tough goal at a time when the cheapest computers cost $100,000 and were locked behind glass. Letwin taught himself FORTRAN from a manual, without access to a computer to try it on. While in high school he took some courses at a small private technical college to get free time on the school's computer.

Letwin studied physics at Purdue, but found he was more interested in getting computer time than in studying physics. Whenever he could, he'd spend his day breaking through the computer's protective environments instead of studying physics. "Saying that there's something that I can't understand, or can't access because I'm not initiated into some mystery is like waving a flag in front of a bull," he says. He finally solved this conflict by changing his major to computer science.

A large part of the enjoyment in computing for Letwin is getting immediate results. He likens computers to pinball machines. "You get positive and negative reinforcement right away." And, being "innately lazy," he likes the idea of computers doing work for him.

Upon getting his Bachelor's and Master's degrees in computer science from Purdue, Letwin went to work for Wintek, a company run by one of his professors. He ended up bidding on a contract to supply Heathkit with a BASIC interpreter, assembler, and editor.

He later moved to Michigan and went to work at Heath but found he didn't agree with its management's direction. "I wasn't happy with the direction of the company, which was run by 'suits' who didn't know computers well but insisted on making the technical decisions anyway." When his concerns with Heath's approach at that time weren't improving, he signed on with Microsoft, then in Albuquerque, New Mexico. "I came here because of Bill Gates," says Letwin, who met Microsoft's chairman during sales meetings at Heath.

Letwin became the first new employee when the firm relocated to Seattle about eight years ago. He started working on a BASIC compiler, then a Pascal compiler.

When he became tired of being a "compiler compiler," he took advantage of his operating system experience to switch to the newly formed operating systems group, starting work on the IBM PC. "We thought the 8086 chip was the first processor powerful enough for general purpose desktop use. We had a vision of an office of the future. We knew that to realize this vision, we'd need a powerful operating system, much more sophisticated then those which were available at that time. Since the only way to insure its existence was to write it ourselves, we got into the operating system business." The widespread acceptance of MS-DOS makes that statement Microsoft's platform for its development of the single-user office automation operating system.

stop the technique anyway, so we tried another approach: mode switching.

The 286 chip switches from real mode to protected mode very easily, but it was not designed to switch back from protected mode to real mode at all. You could only switch back to real mode by causing a full system reset. The idea of doing a full system reset while running a multitasking system was very radical. IBM was aware of the mode switching technique itself and had the capability for it in the AT BIOS for use during power-up diagnostics and for special block copies from high memory. However, this feature was never intended for use during full system operation. What we were thinking of was similar to switching gears in a car by turning the motor off. The idea was to master-clear the processor and restart it. The question was, would this interfere with DMA transfers that might be happening at the same time, lose pending interrupts, or fail in some other way? The IBM ROM code was designed to "switch gears by turning off the car" with the car standing still. We wanted to switch gears with the car traveling at highway speeds, and without causing a jolt. Working together with IBM, we set up an experiment to see if it would work, and it did. However, it took us months to convince everyone involved that the technique was feasible. The first time we presented the idea to IBM Boca, they gave us a funny look and suggested that perhaps we'd had a bit too much sun.

WHAT WE WERE THINKING OF WAS SIMILAR TO SWITCHING GEARS IN A CAR BY TURNING THE MOTOR OFF. THE IDEA WAS TO MASTER-CLEAR THE PROCESSOR AND THEN RESTART IT.

There is one difficulty with mode switching, however. Because the designers thought that it would be an infrequent operation, done primarily during the power-on diagnostics, they implemented it in a fashion that saved some components but produced a very slow mode switch: typically 1,000 microseconds, round trip. During this 1 millisecond interval the system cannot respond to interrupts. Any interrupts that arrive then are processed after the mode switch. This delay would normally be a problem only for high-speed communications at 96K bps or faster while you are simultaneously interacting with a real-mode program. You may experience data overruns while the real-mode screen group is on the display. Mode switching will affect any program that requires a very high-speed interrupt service rate faster than 500 Hz. If the communications protocols on your system can handle missing characters by asking for a retransmit, and most can, then communications will be okay.

MSJ: Why is protection so important in OS/2?

Letwin: We knew we needed a multitasking system to accomplish our goals, and we knew we needed to have it protected. If you run a multiprocessor system, software can accidentally read or change other programs or data if the memory is unprotected. You can't stop it and you can't tell it happened. The system can crash or you could corrupt data. Even in an unprotected DOS 3 system you can have problems with running programs simultaneously. *SideKick*, for example, loads into memory and terminates itself. It keeps memory in use, but the system doesn't know what is in that memory location. If you load several such programs at the same time and they conflict on how they use memory or interrupt vectors, the software will not work correctly, or the system will crash. So in an unprotected environment the system is always vulnerable to flaky software or conflicts between software products from different vendors.

Of even greater importance than this is the fact that we needed to use protected mode to allow programs to access more than 640K bytes of memory. The 640K limit in earlier versions of DOS is a result of the addressing limitations of real mode. Protected mode programs are released from this restriction, and can use many megabytes of memory. As memory costs continue to drop and memory consumption by applications rises, it has become unacceptable to be restricted to the 640K that real mode provides.

MSJ: How did you plan such an ambitious project, both conceptually and in getting the needed resources?

Letwin: The first step was to decide what we wanted the system to do. Then, when we discovered how many of those capabilities we could implement, and by when, we could revise our goals. As a result, the initial planning process was iterative. We needed to

have multitasking and to execute existing DOS 3 programs. While there were still only a few engineers on the project, we thought about techniques such as mode switching and when necessary, prototyped some code. As we got a better handle on what we could do, and how long it would take, we met again with Bill Gates and revised our goals. The tone of the first phase was as ambitious as possible, and we listed all the things we thought we could do. The second phase involved determining which of those features were feasible in the required time frame. As soon as the general scope of the project was determined, the engineers met in groups of two and three to lay out further details of the architecture of the system.

After the initial meetings came the joint design work. Engineering then finalized the goals. Because of the scope of the project, this design phase took a few months.

Next was the sign-up phase, a tradition in the computer industry, where we explained the excitement, the importance, and the tight schedule of the project, then let everyone "sign up" for a long stretch of hard, focused work. You can't force someone to excel; you've got to provide the opportunity and the encouragement and then let them excel.

Then the actual work began. We decided on the key systems and the key characteristics of the product. Next came the design sessions for each of the key systems. About a half dozen people were involved, typically three from Microsoft and three from IBM.

During the coding months, my job was to understand fully all of the parts of the system and to make sure that everthing worked toward achieving our overall system goals. In a project this large, with this many people working on it, it's easy for design elements to diverge. One project member might think that speed is more important than code size, and write his code that way. Another might think the opposite and write his code the other way. The result would be that neither goal is achieved. I participated in the design of each component to insure that the goals and design philosophies of the system's architecture were consistent throughout the code. On most projects, this role can be fulfilled on a "part time" basis, leaving time for the chief designer to also get to write some code. Unfortunately, on the OS/2 project this position required a full time effort, leaving me little opportunity to write any real code.

MSJ: How did you divide up tasks on the project?

Letwin: The most efficient programming is done by one person. It's a classic saying in the industry that "if one person can do it in one year, two people can do it in two years." But in a project like OS/2, although one person might be able to do it in 8 man years, you can't afford to wait 8 years, so you have to make it a multi-person project and have it take 30 man years. In order to reduce the inefficiency introduced by having multiple people on a project, it's important to break the work up properly, and to have each piece of work be a one man project. This project had about 35 people on it.

The programmers worked on individual parts of the code, but they were organized into teams with a maximum of eight people each to facilitate coordinating their work. There was a test team, two teams of developers each headed by a senior engineer, a build team with four people, a test team with five people, and myself as the chief architect. This project was so complicated, especially since development work was simultaneously taking place at IBM in Boca Raton, Florida, that we needed a build team just to coordinate the sources, build the executable binaries, and otherwise bring organization to the physical aspects of software development. The test team, as its name implies, was responsible for the development of test cases, bug tracking, and performance measurement.

MSJ: What was the most difficult part of managing the project?

Letwin: This project's very large staff added to the complexity of integrating changes made at the IBM locations and timing the development between the two sites. We had to be especially careful that any changes made didn't affect other parts of the project. In the source code control system, for example, we split the source into about 200 modules, with modification tracking on each. We had a rule that you couldn't add a change to a module until

DURING THE CODING MONTHS, MY JOB WAS TO UNDERSTAND ALL THE PARTS OF THE SYSTEM AND TO MAKE SURE THAT EVERYTHING WORKED TOWARD ACHIEVING OUR OVERALL SYSTEM GOALS.

you'd proven that the change wouldn't break anything. That way we retained a totally stable version. Without this rule, each engineer could have spent days looking for problems that were, in fact, introduced by another engineer.

COMPUTER PROGRAMMING IS 20 PERCENT SCIENCE, 60 PERCENT ENGINEERING, AND 20 PERCENT ART. THE "ART" ELEMENT IS THE HARDEST TO ACHIEVE, AND IT'S THE HARDEST TO EXPLAIN.

MSJ: IBM and Microsoft, to put it mildly, have very different corporate cultures and philosophies. How did you mesh the two environments during the development of OS/2?

Letwin: First, whenever we arrived in Boca Raton, we rushed into the bathroom and changed into suits.

But seriously, there were some times when differences of opinions were frustrating. Normally each company builds its own products and calls all the shots. But in a joint development, compromises were necessary. IBM engineers were slowed down by a set of rules which are intended to bring stability to large IBM projects. This naturally caused some conflicts. IBM-ers undoubtedly felt that Microsoft engineers were "wild and crazy guys."

MSJ: What parts of OS/2 would you do differently if you had the choice?

Letwin: A year ago it became clear we weren't going to meet our schedules, and we had to go through a simplification procedure. For example, we took out named pipes, a form of inter-process communication to send information to another program (now part of the OS/2 LAN Manager). We'll put that in another release of the product. Installable file systems were another victim. As a result, the hard disk file system still doesn't perform as well as we'd all like. These features will be added later. Also, we now know the parts of the system which turned out to be bigger or slower than they should, or which in some other way were a bit disappointing. Naturally we'd give them additional attention. At the end of any project, there are always a few things you'd like to improve. And we will, in the next release.

MSJ: How does it feel to give up your product when it's ready to go to market?

Letwin: At first, I felt relief when we got the product through testing and it went out the door. The last 20 percent is hard work. You just get tired of the product. The exciting stuff takes place in the earlier stages of the project. The last days are dealing with problem after problem—fixing bugs, making it smaller, making it faster, etc. You've worked on it so much there dosen't seem to be much change over the last few months. You're tired and all you do is deal with problems. In my opinion, it's the ability to keep working hard during the last part of a long project that separates the great people from the average. In any case, when you finally move on to something else, you feel a mixture of relief that it's completed, excitement about your new project, and anxiety about how well your efforts will be received.

MSJ: What's your philosophy of programming?

Letwin: Software is special at our level. Computer programming is 20 percent science, 60 percent engineering, and 20 percent art. The "art" element is the hardest to achieve, and it's the hardest to explain. When engineers talk about the "elegance" of their work, people usually think that such elegance is non-functional: the engineers are just indulging themselves. This might be the case in mechanical or civil engineering since an elegant bridge carrys no more load than an inelegant one. But it isn't so in advanced software development because mechanical engineering deals with physical objects and software engineering deals with information. Physical objects have properties independent of human perception, but information doesn't. In other words, although a bridge is equally functional whether or not it's elegant, information is much more useful when it's organized in a clear, flexible and useable form—a form a programmer would call "elegant." Bridge design is organized to meet just two goals—buildability and final function. Software is designed to meet those goals plus one more. It has to have minimal intellectual complexity and maximum flexibility so that it can be understood by the programmers who will maintain and upgrade it in the future. ❑

A Simple Windows Application for Custom Color Mixing

Charles Petzold

COLORSCR ("Color Scroll"), is a program for *Microsoft Windows* that displays three scroll bars in the left half of the client area, which are labeled "Red," "Green," and "Blue," as shown in **Figure 1**. As you scroll the scroll bars, the right half of the client area will change to the composite color that is produced by the mix of the three primary colors. The numeric values of the three primary colors are displayed under the three scroll bars.

You can scroll the scroll bars by using either the mouse or the keyboard. COLORSCR works best when *Windows* is installed to use a color display, such as an Enhanced Graphics Adapter connected to a color monitor, but you can also run the program on a monochrome display.

You can use COLORSCR as a development tool in experimenting with color and choosing attractive colors for your own *Windows* programs. But COLORSCR is most interesting for what it *doesn't* do.

For instance, if you've done some *Windows* programming, you'll probably assume that the Red, Green, and Blue labels on top of the scroll bars and the color values at the bottom are displayed to the client area by using the TextOut function.

However, COLORSCR does not use TextOut.

Also, you'll probably assume that the colored rectangle at the right half of the client area is drawn by using the Rectangle or FillRect function.

No, COLORSCR doesn't use these functions either, or any of the other GDI drawing functions.

When COLORSCR is made into an icon, the entire surface of the icon is painted with the selected color. You might think that COLORSCR traps WM_SIZE messages to check if it's being iconed and then paints the client area differently if it is.

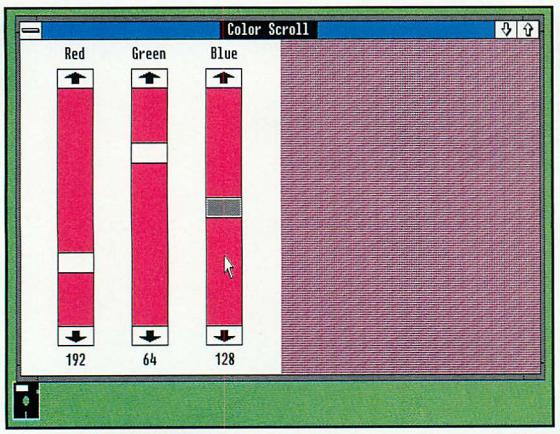
Wrong again. COLORSCR does not know when it's an icon and doesn't have separate icon logic.

In fact, COLORSCR doesn't process WM_PAINT messages and doesn't directly display *anything* to its client area. At some point the COLORSCR program does obtain a handle to a display context, but only for the purpose of determining the height of a character in the default system font.

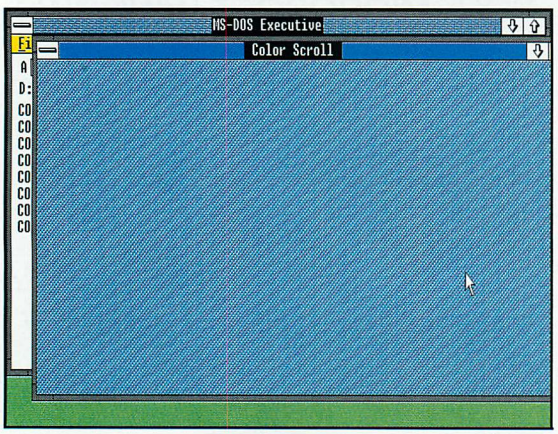
In short, COLORSCR is one of the laziest *Windows* pro-

Charles Petzold is a writer and columnist for several computer magazines.

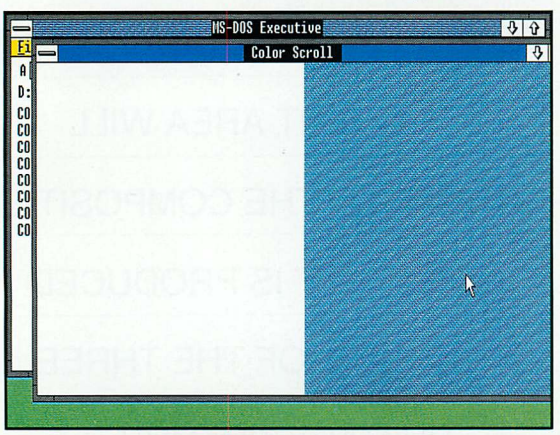
COLORSCR IS A PROGRAM FOR *MICROSOFT WINDOWS* THAT DISPLAYS THREE SCROLL BARS IN THE LEFT HAND OF THE CLIENT AREA, WHICH ARE LABELED "RED," "GREEN," AND "BLUE." AS YOU SCROLL THE SCROLL BARS, THE RIGHT HALF OF THE CLIENT AREA WILL CHANGE TO THE COMPOSITE COLOR THAT IS PRODUCED BY THE MIX OF THE THREE PRIMARY COLORS.



▲ **Figure 1** The COLORSCR program for *Microsoft Windows* displays a color on the right half of the client area based on the settings of three scroll bars on the left.



▲ **Figure 2** The background of COLORSCR's client area is the composite color.



▲ **Figure 3** The left half of the client area is blocked out by an SS_WHITERECT child window.

grams ever. How can it afford to do so little? Because COLORSCR puts its children to work.

Child Windows

Child windows are separate windows that can be overlaid on the client area of a tiled or pop-up window. They usually have their own window functions separate from the parent window function. In their simplest form, child windows can simply divide the client area of a window into smaller rectangular windows, each with its own window function.

For instance, the *Windows* MS-DOS Executive uses three child windows to divide its client area into sections for the drive icons, the current subdirectory path, and the file list. Three different window functions process messages from the three child windows. This simplifies painting and mouse processing in the MS-DOS Executive because each child window has its own client area coordinates.

One type of child window is called a *child window control*, which usually sends messages back to its parent window (the tiled or pop-up window on which the child control appears) based on mouse or keyboard input to the control.

Windows contains several predefined child window controls, which take the form of buttons, check boxes, edit boxes, list boxes, text strings, and scroll bars. Child window controls are most often used in dialog boxes; in such instances, the placement and size of the child window controls are defined in a dialog box template contained in the program's resource script. When you create a dialog box template, you can either calculate the position and sizes manually or you can use the DIALOG program included with the *Windows Software Development Kit 1.03* (see "Latest Dialog Editor Speeds *Windows* Applications Development," *Microsoft Systems Journal*, Vol. 1 No. 1, p. 25).

However, you can also use predefined child window controls on the surface of a tiled or pop-up window's client area. You create each child window with a CreateWindow call and adjust the position and size of the child windows with calls to MoveWindow.

COLORSCR uses ten predefined child window controls. The color that appears on the right half of the window is actually the background color of the tiled window, as shown in **Figure 2**.

On the left half of the client area there is a "static" child window called SS_WHITERECT (white rectangle) that blocks out half the client area (see **Figure 3**). The three scroll bars are child window controls with the style SBS_VERT (see **Figure 4**) placed on top of the SS_WHITERECT child. Six more static child windows of style SS_CENTER (centered text) provide the labels and the color values (see **Figure 5**).

Raising the Children

The COLORSCR.C source code, the COLORSCR.DEF module definition file, and the COLORSCR make-file are shown in **Figures 6, 7,** and **8,** respectively. If you have the *Microsoft C Compiler 4.0* and the *Windows Software Development Kit* installed on your hard disk, you will be able to create COLORSCR.EXE by executing:

```
MAKE COLORSCR
```

COLORSCR creates its normal tiled window and the ten child windows within the WinMain function through the use of the *Windows* CreateWindow call.

The first parameter to the CreateWindow call indicates the window class of the window. The tiled window uses the window class called "ColorScr," which is defined and registered during WinMain initialization; this is perfectly normal. The child windows use classes called "static" (for the white rectangle and the text) and "scrollbar," which are the predefined child window classes. The window functions for these child windows are contained within *Windows*.

For a tiled window, the second parameter to the CreateWindow call is the text that appears in the window's caption bar. For the six child windows that display text, this parameter is the text displayed in the child window. For the rectangle and scroll bars, the text is ignored.

The window style parameter for all ten child windows includes the WS_CHILD indentifier. The window style for the three scroll bars includes the SBS_VERT identifier, indicating a vertical scroll bar. For the static child windows, the SS_CENTER identifier indicates centered text, and the SS_WHITERECT identifier indicates a white rectangle.

The fourth through seventh parameters to the CreateWindow call normally indicate a position and size of the child window. The position is relative to the upper left-hand corner of the client area of the parent window. These values are all set to 0 initially because the placement and sizing depend on the size of the client area, which is not yet known.

For child windows, the eighth parameter is the window handle of the parent, which is simply the hWnd value returned from the first CreateWindow call. The following parameter is a *child ID* that uniquely identifies the child window. We'll use this child ID later on.

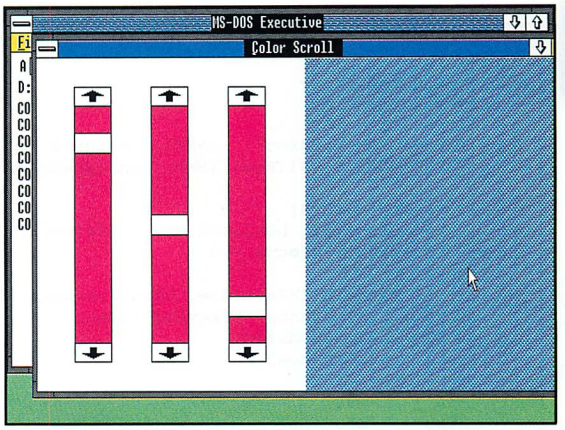
COLORSCR's window function (WndProc) resizes all ten child windows when it receives a WM_SIZE message. The formulas are based upon xClient, the width of the client area, yClient, the height of the client area, and yChar, the height of a character.

Whenever you resize the COLORSCR window, the sizes of the child windows change proportionally. This is particularly interesting for the scroll bars. Depending on the dimensions of COLORSCR's window, the scroll bars can be long and thin or short and stubby (see **Figure 9**). They may look a little peculiar, but they work just fine.

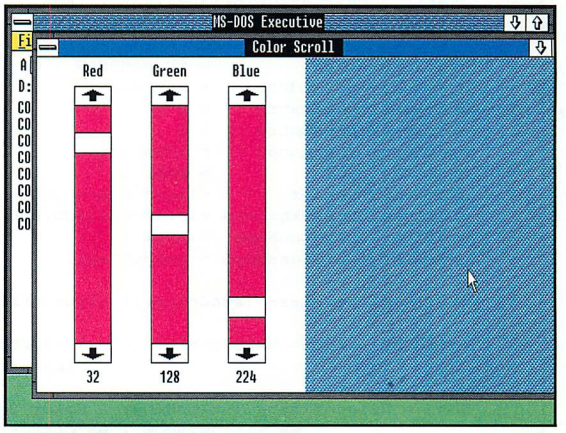
Scroll Bar Messages

Child window controls that take the form of buttons, edit boxes, list boxes, and scroll bars pass messages back to the window function of the parent window. Buttons, edit boxes, and list boxes will post WM_COMMAND messages to the parent window. Scroll bar controls, however, will use the messages WM_VSCROLL (to scroll vertically) and WM_HSCROLL (for scrolling horizontally) to indicate that the scroll bar has been clicked with the mouse or the scroll bar thumb is being dragged.

The first step in using the scroll bars is to set a range for



▲ **Figure 4** **The scroll bars are SBS_VERT child window controls.**



▲ **Figure 5** **The six text labels are six SS_CENTER child windows.**

WHENEVER YOU RESIZE THE COLORSCR WINDOW, THE SIZES OF THE CHILD WINDOWS CHANGE PROPORTIONALLY. THIS IS PARTICULARLY INTERESTING FOR THE SCROLL BARS. DEPENDING ON THE DIMENSIONS OF COLORSCR'S WINDOW, THE SCROLL BARS CAN BE LONG AND THIN OR SHORT AND STUBBY.

```
/* COLORSCR.C -- Color Scroll (using child window controls) */

#include <windows.h>
#include <stdlib.h>

long FAR PASCAL WndProc    (HWND, unsigned, WORD, LONG) ;
long FAR PASCAL ScrollProc (HWND, unsigned, WORD, LONG) ;

FARPROC lpfnOldScr[3] ;
HWND    hChScrol[3], hChLabel[3], hChValue[3], hChRect ;
short   color[3], nFocus = 0;

int PASCAL WinMain (hInstance, hPrevInstance, lpszCmdLine, nCmdShow)
     HANDLE   hInstance, hPrevInstance;
     LPSTR    lpszCmdLine;
     int      nCmdShow;
     {
     MSG      msg;
     HWND     hWnd ;
     WNDCLASS wndclass ;
     FARPROC  lpfnScrollProc ;
     short    n ;
     static   char *szColorLabel[] = { "Red", "Green", "Blue" } ;
     static   char szAppName[] = "ColorScr" ;-

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = NULL ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = CreateSolidBrush (0L) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass)) return FALSE;

     lpfnScrollProc = MakeProcInstance ((FARPROC) ScrollProc,
                                        hInstance) ;

     hWnd = CreateWindow (szAppName, " Color Scroll ",
                    WS_TILEDWINDOW | WS_CLIPCHILDREN,
                    0, 0, 0, 0, NULL, NULL, hInstance, NULL) ;

     hChRect = CreateWindow ("static", NULL,
                    WS_CHILD | WS_VISIBLE | SS_WHITERECT,
                    0, 0, 0, 0, hWnd, 9, hInstance, NULL) ;

     for (n = 0 ; n < 3 ; n++)
          {
          hChScrol[n] = CreateWindow ("scrollbar", NULL,
                         WS_CHILD | WS_VISIBLE | WS_TABSTOP | SBS_VERT,
                         0, 0, 0, 0, hWnd, n, hInstance, NULL) ;

          hChLabel[n] = CreateWindow ("static", szColorLabel[n],
                         WS_CHILD | WS_VISIBLE | SS_CENTER,
                         0, 0, 0, 0, hWnd, n + 3, hInstance, NULL) ;

          hChValue[n] = CreateWindow ("static", "0",
                         WS_CHILD | WS_VISIBLE | SS_CENTER,
                         0, 0, 0, 0, hWnd, n + 6, hInstance, NULL) ;

          lpfnOldScr[n] = (FARPROC) GetWindowLong (hChScrol[n],
                            GWL_WNDPROC) ;
          SetWindowLong (hChScrol[n], GWL_WNDPROC,
                            (LONG) lpfnScrollProc) ;

          SetScrollRange (hChScrol[n], SB_CTL, 0, 255, FALSE) ;
          SetScrollPos   (hChScrol[n], SB_CTL, 0, FALSE) ;
          }

     ShowWindow (hWnd, nCmdShow) ;
     UpdateWindow (hWnd);
```

CONTINUED

▲ **Figure 6** COLORSCR.C source code for the Color Scroll program.

the scroll bar, which is done in WinMain. Since the primary color values range from 0 to 255, the range is set accordingly:

```
SetScrollRange
  (hChScrol[n],
  SB_CTL,
  0, 255, FALSE) ;
```

When the WndProc window function receives a WM_VSCROLL message, the high word of the lParam parameter is the child window ID number, which is the number specified in the ninth parameter to the CreateWindow call. For the three scroll bars, we have conveniently set that ID to a 0, 1, and 2. That tells WndProc which scroll bar is generating the message:

```
n = GetWindowWord
       (HIWORD
         (lParam),
       GWW_ID) ;
```

Because the handles to the child windows had been saved in arrays when the windows were created, WndProc can set the new value of the appropriate scroll bar by using the SetScrollPos call:

```
SetScrollPos
  (hChScrol[n],
  SB_CTL,
  color[n],
  TRUE) ;
```

and it can change the text of the child window at the bottom of the scroll bar:

```
SetWindowText
  (hChValue[n],
  itoa
    (color[n],
    szbuffer,
    10)) ;
```

Keyboard Interface

Scroll bar controls can also process keystrokes, but only if they have the input focus. (Because only one window can receive keyboard input at any

time, the window that receives the keyboard input is referred to as the window with the input focus.) The keyboard cursor keys are translated into the scroll bar messages:

Cursor Key	Scroll Bar Message wParam Value
Home	SB_TOP
End	SB_BOTTOM
PgUp	SB_PAGEUP
PgDown	SB_PAGEDOWN
Left-arrow	SB_LINEUP
Up-arrow	SB_LINEUP
Right-arrow	SB_LINEDOWN
Down-arrow	SB_LINEDOWN

In fact, the SB_TOP and SB_BOTTOM scroll messages are never generated when using the mouse on the scroll bar, only when using the keyboard.

The Input Focus

If you want a scroll bar control to obtain the input focus when the scroll bar is clicked with the mouse, you must include the WS_TABSTOP identifier in the window class parameter of the CreateWindow call. When a scroll bar has the input focus, a blinking gray block is displayed on the scroll bar thumb.

However, to provide a full keyboard interface to the scroll bars, some more work is necessary. First, the WndProc window function must specifically give a scroll bar the input focus. It does this by processing the WM_FOCUS message, which the parent tiled window receives when the parent window obtains the input focus. WndProc simply sets the input focus to one of the scroll bars:

```
SetFocus
(hChScrol[nFocus]);
```

But you also need some way to get from one scroll bar to another by using the keyboard, preferably with the Tab key.

Figure 6 CONTINUED

```
 while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage  (&msg) ;
          }
     return msg.wParam ;
     }

long FAR PASCAL WndProc (hWnd, iMessage, wParam, lParam)
     HWND        hWnd;
     unsigned    iMessage;
     WORD        wParam;
     LONG        lParam;
     {
     HDC         hDC ;
     TEXTMETRIC tm ;
     char        szbuffer[10] ;
     short       n, xClient, yClient, yChar ;

     switch (iMessage)
          {
          case WM_SIZE :
               xClient = LOWORD (lParam) ;
               yClient = HIWORD (lParam) ;

               hDC = GetDC (hWnd) ;
               GetTextMetrics (hDC, &tm) ;
               yChar = tm.tmHeight ;
               ReleaseDC (hWnd, hDC) ;

               MoveWindow (hChRect, 0, 0, xClient / 2, yClient, TRUE);

               for (n = 0 ; n < 3 ; n++)
                    {
                    MoveWindow (hChScrol[n],
                         (2 * n + 1) * xClient / 14, 2 * yChar,
                         xClient / 14, yClient - 4 * yChar, TRUE) ;

                    MoveWindow (hChLabel[n],
                         (4 * n + 1) * xClient / 28, yChar / 2,
                         xClient / 7, yChar, TRUE) ;

                    MoveWindow (hChValue[n],
                         (4 * n + 1) * xClient / 28,
                         yClient - 3 * yChar / 2,
                         xClient / 7, yChar, TRUE) ;
                    }
               SetFocus (hWnd) ;
               break ;

          case WM_SETFOCUS:
               SetFocus (hChScrol[nFocus]) ;
               break ;

          case WM_VSCROLL :
               n = GetWindowWord (HIWORD (lParam), GWW_ID) ;

               switch (wParam)
                    {
                    case SB_PAGEDOWN :
                         color[n] += 15 ;          /* fall through */
                    case SB_LINEDOWN :
                         color[n] = min (255, color[n] + 1) ;
                         break ;
                    case SB_PAGEUP :
                         color[n] -= 15 ;          /* fall through */
                    case SB_LINEUP :
                         color[n] = max (0, color[n] - 1) ;
                         break ;
                    case SB_TOP:
                         color[n] = 0 ;
                         break ;
                    case SB_BOTTOM :
                         color[n] = 255 ;
                         break ;
```

CONTINUED

Figure 6 CONTINUED

```
case SB_THUMBPOSITION :
                    case SB_THUMBTRACK :
                         color[n] = LOWORD (lParam) ;
                         break ;
                    default :
                         break ;
                    }
               SetScrollPos   (hChScrol[n], SB_CTL, color[n], TRUE) ;
               SetWindowText  (hChValue[n],
                               itoa (color[n], szbuffer, 10)) ;

               DeleteObject (GetClassWord (hWnd, GCW_HBRBACKGROUND)) ;
               SetClassWord (hWnd, GCW_HBRBACKGROUND, CreateSolidBrush
                            (RGB (color[0], color[1], color[2]))) ;

               InvalidateRect (hWnd, NULL, TRUE) ;
               break ;

          case WM_DESTROY:
               DeleteObject (GetClassWord (hWnd, GCW_HBRBACKGROUND)) ;
               PostQuitMessage (0) ;
               break ;

          default :
               return DefWindowProc (hWnd, iMessage, wParam, lParam) ;
          }
     return 0L ;
     }

long FAR PASCAL ScrollProc (hWnd, iMessage, wParam, lParam)
     HWND       hWnd ;
     unsigned   iMessage ;
     WORD       wParam ;
     LONG       lParam ;
     {
     short      n = GetWindowWord (hWnd, GWW_ID) ;

     switch (iMessage)
          {
          case WM_KEYDOWN:
               if (wParam == VK_TAB)
                    SetFocus (hChScrol[(n +
                         (GetKeyState (VK_SHIFT) < 0 ? 2 : 1)) % 3]) ;
               break ;

          case WM_SETFOCUS:
               nFocus = n ;
               break ;
          }
     return CallWindowProc (lpfnOldScr[n], hWnd, iMessage, wParam,
                            lParam) ;
     }
```

This is more difficult, because once a scroll bar has the input focus it processes all keystrokes. The scroll bar only cares about the cursor keys and ignores the Tab key.

Window Subclassing

Adding a facility to COLORSCR in order to jump from one scroll bar to another using the Tab key requires *window subclassing*.

The window function for the scroll bar controls is somewhere inside *Windows*, but the address of this window function can be obtained by a call to GetWindowLong using the GWL_WNDPROC identifier as a parameter. Moreover, you can set a new window function for the scroll bars by calling SetWindowLong.

This facility allows you to hook into the existing scroll bar window function, process some messages within your own program, and pass all the other messages to the old window function.

The window function that does preliminary scroll bar message processing in COLORSCR is called ScrollProc and is found toward the end of the COLORSCR.C listing. Since ScrollProc is a function within COLORSCR that is called by *Microsoft Windows*, it must be defined as FAR PASCAL and it must be listed in the EXPORTS in the COLORSCR.DEF module definition file.

First, to ensure that ScrollProc accesses the proper data segment, you must obtain a far address for the function with MakeProcInstance:

```
lpfnScrollProc =
  MakeProcInstance
    ((FARPROC)
      ScrollProc,
    hInstance) ;
```

For each of the three scroll bars, COLORSCR uses GetWindowLong to obtain and save the address of the existing scroll bar window function:

```
lpfnOldScr[n] =
  (FARPROC)
    GetWindowLong
      (hChScrol[n],
      GWL_WNDPROC) ;
```

Next, it sets the new scroll bar window function:

```
SetWindowLong
  (hChScrol[n],
```

```
    GWL_WNDPROC,
    (LONG)
      lpfnScrolProc) ;
```

Now, the function ScrollProc gets first dibs on all messages that *Windows* sends to the scroll bar window function for the three scroll bars in COLORSCR (but not, of course, scroll bars in other programs). The ScrollProc window function simply changes the input focus to the next, or previous, scroll bar when it receives a Tab or Shift-Tab keystroke. It calls the old scroll bar window function with CallWindowProc.

Use of Color

The *Windows* functions that require (or return) a color value use an unsigned long (32-bit) integer, where the lowest three bytes specify red, green, and blue values ranging from 0 through 255 (see **Figure 10**).

This results in a potential 2^{24} (about 16 million) possible colors.

This unsigned long is often referred to in the *Windows* documentation as rgbColor. The WINDOWS.H header file provides several macros for working with rgbColor values. The RGB macro in WINDOWS.H takes three arguments representing red, green, and blue values and then sticks the bytes together in order to form an unsigned long:

```
#define RGB(r,g,b)
  (((DWORD)
      (b << 8 | g)
      << 8)
      | r)
```

Thus, the value

```
RGB (255, 0, 255)
```

is really 0x00FF00FF, an rgbColor value for magenta. When all three arguments are set to 0, the rgbColor is black; all three arguments set to 255 yields white.

C mavens will justifiably recoil upon seeing this RGB macro. The b, g, and r identifiers within the definition should be surrounded by parentheses to avoid errors when one of the parameters is an expression involving operators of higher precedence than << and |. You should watch out when you use expressions for the b, g, and r values.

The GetRValue, GetGValue, and GetBValue macros extract the unsigned character primary color values from an unsigned long rgbColor value. These macros are not used in COLORSCR, but they sometimes come in handy when you are using *Windows* functions that will return rgbColor values to your program.

The *Windows* display driver for the Enhanced Graphics Adapter attached to a color display uses only three of the four color planes of the EGA board, so only eight pure colors are possible. (In COLORSCR, the eight pure colors result when all scroll bars are either at the top or bottom positions.)

```
NAME            ColorScr
DESCRIPTION     'Color Scroll by Charles Petzold, 1987'
STUB            'WINSTUB.EXE'
CODE            MOVABLE
DATA            MOVABLE MULTIPLE
HEAPSIZE        1024
STACKSIZA       4096
EXPORTS         WndProc
                ScrollProc
```

▲ **Figure 7** **COLORSCR.DEF module definition file for the Color Scroll program.**

```
colorscr.obj : colorscr.c
   cl -c -d -Gsw -Os -W2 -Zpd colorscr.c

colorscr.exe : colorscr.obj colorscr.def
   link4 colorscr, /align:16, /map /line, slibw, colorscr
   mapsym colorscr
```

▲ **Figure 8** **COLORSCR make-file for the Color Scroll program.**

THE FUNCTION SCROLLPROC GETS FIRST DIBS ON ALL MESSAGES THAT *WINDOWS* SENDS TO THE SCROLL BAR WINDOW FUNCTION FOR THE THREE SCROLL BARS IN COLORSCR (BUT NOT, OF COURSE, SCROLL BARS IN OTHER PROGRAMS). THE SCROLLPROC WINDOW FUNCTIONS SIMPLY CHANGE THE INPUT FOCUS TO THE NEXT, OR PREVIOUS, SCROLL BAR WHEN IT RECEIVES A TAB OR SHIFT-TAB KEYSTROKE. IT CALLS THE OLD SCROLL BAR FUNCTION WITH CALLWINDOWPROC.

Windows can display additional colors on the EGA by *dithering*, the creation of a pixel pattern that combines pixels of different pure colors. You'll find from experimenting on an EGA that the dithering pattern remains the same for every increment of four units for the red, green, and blue values. So, for an EGA, you have 2^{18} or 262,144 dithered colors.

On a monochrome display—such as a CGA, an EGA connected to a monochrome display, or a Hercules board—*Windows* uses 64 different patterns of white and black. The density of white is approximately equal to

$$\frac{\text{red} + \text{green} + \text{blue}}{255 * 3}$$

where the red, green, and blue values range from 0 to 255.

Watch out for this—if you're developing using a color display, you might see nothing wrong with putting a pure red object against a pure blue background. However, on a monochrome display the pure red and pure blue "colors" appear to be the same.

Background Color

When COLORSCR is first executed, it sets up a window class structure, called wndclass, that defines certain characteristics of its window. One member of this structure is a handle to a brush that *Windows* uses to color the background of the client area. This starts off as a solid black brush:

```
wndclass.hbrBackground
=CreateSolidBrush(0L);
```

CreateSolidBrush requires an rgbValue and returns a handle to a brush. Although the hbrBackground member of the structure refers to a background, in more technical terms it is the brush that *Windows* uses to erase the window's client area in preparation for repainting by the program's window function.

When you change the settings of COLORSCR's scroll bars, the program must create a new brush and put the new brush handle in the window class structure. Just as we were able to get and set the scroll bar window function by using GetWindowLong and SetWindowLong, we can get and set the handle to this brush by using the calls GetClassWord and SetClassWord.

First, it's necessary to delete the existing brush:

```
DeleteObject
  (GetClassWord
    (hWnd,
    GCW_HBRBACKGROUND
    )
  ) ;
```

Then the new brush can be created and the handle inserted in the window class structure:

```
SetClassWord
  (hWnd,
  GCW_HBRBACKGROUND,
  CreateSolidBrush
    (RGB
      (color[0],
      color[1],
      color[2]))) ;
```

Now, the next time *Windows* recolors the background of the window, *Windows* will use this new brush. To force *Windows* to erase the background, we invalidate the entire client area:

```
InvalidateRect
  (hWnd, NULL, TRUE);
```

The TRUE (nonzero) value as the third parameter indicates that we want the background erased before repainting.

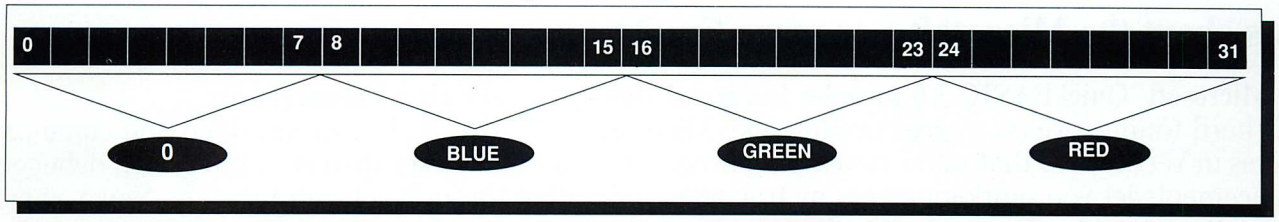
InvalidateRect causes *Windows* to put a WM_PAINT message in the message queue of the window function. Because WM_PAINT messages are low priority, this message will not be processed immediately if you are still moving the scroll bar with the mouse or the cursor keys. Alternatively, if you want the window to be updated immediately after the color is changed, you could add the statement

```
UpdateWindow(hWnd) ;
```

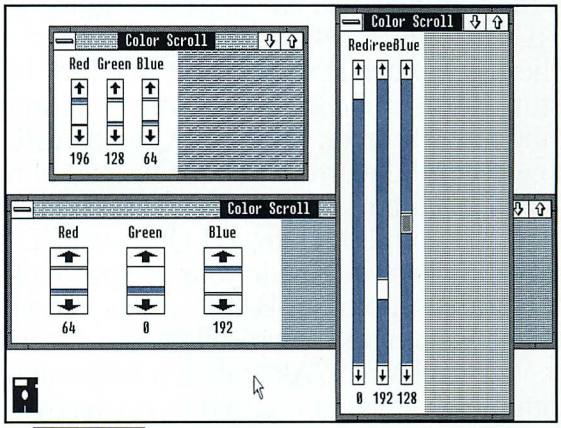
after the InvalidateRect call. However, this slows down keyboard and mouse processing.

COLORSCR's WndProc function doesn't process the WM_PAINT message and just passes it on to DefWindowProc. *Windows*'s default processing of WM_PAINT messages simply involves calling BeginPaint and EndPaint to validate the window. Because we specified in the InvalidateRect call that the background should be erased first, the BeginPaint call causes *Windows* to generate the message WM_ERASEBKGND (erase background). WndProc ignores this message also. *Windows* processes it by erasing the background of the client area with the brush specified in the window class.

Normally *Windows* would erase the entire client area using the window class brush. This would cover the 10 child windows. *Windows* would then have to send WM_PAINT messages to all 10 child windows so that they could repaint themselves, which would be very annoying. However, you can avoid this problem by using the WS_CLIPCHILDREN style value when you are first creating the parent window with the CreateWindow call. The WS_CLIPCHILDREN style prevents the parent



▲ **Figure 10** An unsigned long (32-bit) integer where the lowest three bytes specify red, green, and blue values.



▲ **Figure 9** The size and position of the child windows are based on the size of the window. Note the various shapes of the scroll bars.

window from painting over its children. If you then take the WS_CLIPCHILDREN style out of CreateWindow, you'll see a big difference in how COLORSCR works.

Like all GDI objects, the brushes created by a program that uses CreateSolidBrush are not automatically deleted by *Windows* when the program terminates. So far, we've been good about deleting each brush before creating a new one, but when the program is about to terminate, there is still one final brush in the window class that we should discard. Thus, during the processing of the WM_DESTROY message, DeleteObject is called once more:

```
DeleteObject
  (GetClassWord
    (hWnd,
    GCW_HBRBACKGROUND
    )
) ;
```

Multiple Instances

Normally, most *Windows* programs will reuse the same window class when you load multiple instances of the program. The window class is registered only if the previous instance is NULL:

```
if (!hPrevInstance)
  {wndclass.style =
    CS_HREDRAW |
    CS_VREDRAW ;
      .
      .
      .
  }
```

But COLORSCR must take a different approach because the background color is specified in the window class. If all instances of COLORSCR used the same window class, then each instance would alter and use the same background color.

This problem is easy to solve: each instance simply registers its own window class. COLORSCR doesn't check if it's the first instance or the twentieth. Contrary to *Windows* documentation, a window class registered with the same name as a previously registered window class does not replace the earlier window class. A program simply has the option of using its own window class or a previously registered window class. COLORSCR takes the former approach.

COLORSCR as an Icon

One final mystery remains. When you make COLORSCR into an icon, the entire surface of the icon appears as the selected color rather than just the right half. Yet COLORSCR doesn't seem to have any separate icon logic.

Notice that COLORSCR specifies a NULL icon in the window class:

```
wndclass.hIcon=NULL;
```

This indicates that COLORSCR is responsible for painting its icon. But why does the entire icon appear as the selected color?

This is the simplest part of COLORSCR—*Windows* hides child windows when a program is iconed. The colored background is then completely uncovered. So, when the parent is sleeping, the children are neither seen nor heard. ❑

IF ALL INSTANCES OF COLORSCR USED THE SAME WINDOW CLASS, THEN EACH INSTANCE WOULD ALTER AND USE THE SAME BACKGROUND COLOR. THE PROBLEM IS EASY TO SOLVE: EACH INSTANCE SIMPLY REGISTERS ITS OWN WINDOW CLASS. COLORSCR DOESN'T CHECK IF ITS THE FIRST INSTANCE OR THE TWENTIETH.

MICROSOFT LANGUAGES NEWSLETTER VOL. 2, NO. 6

News about the Microsoft Language Family

New Microsoft® QuickBASIC 3.0 Includes Integrated Debugger and Editor Enhancements

It's hard to improve on a great product like Microsoft QuickBASIC, but we've added and enhanced features in Version 3.0 that make developing programs faster and easier than ever. Integrated debugger enhancements let you pinpoint errors by tracing or animating through your source code. Now you can observe the contents of your program's variables as the program is running. And set dynamic breakpoints at runtime to easily stop the program where you want. You don't have to add STOP statements to your programs and wait for another compilation anymore. With the debugger completely integrated into the compiler, you can start debugging your program while it's running simply by pressing CTRL-BREAK.

Microsoft Quick-BASIC's built-in editor has been enhanced to support overtype as well as the existing insert mode so editing your programs is easier. In contrast to other compilers that give up after finding a single error, Microsoft QuickBASIC keeps track of all errors found during compilation. You can correct all the errors at once instead of recompiling for each error. And the Microsoft QuickBASIC Version 3.0 editor is compatible with SuperKey,® ProKey,™ and SideKick® programs.

As in Version 2.0, Microsoft QuickBASIC 3.0 supports multiple-module programming. Separate compilation lets you divide your program into pieces that are compiled independently. These pieces can be integrated into other BASIC programs without recompilation. The linker in Microsoft QuickBASIC makes it easy to add Microsoft Macro Assembler object files to your BASIC programs. Just assemble your Microsoft Macro Assembler routines and let the linker incorporate them into Microsoft QuickBASIC automatically. There's no need to convert your Microsoft Macro Assembler routines into COM files or write them as hexadecimal numbers in your BASIC source file as in other compilers.

Math Coprocessor Support in Microsoft QuickBASIC 3.0 Speeds Programs

Microsoft QuickBASIC now has support for the 8087 and 80287 math coprocessors. The full 80-bit IEEE math support of the math coprocessor is needed for programs that demand the most precise calculations. In addition to this, you may use the existing Microsoft Binary Math support for fast 64-bit math or the new 8087 emulation routines for more accuracy when you don't have a coprocessor.

There have also been many dramatic speed enhancements in Microsoft QuickBASIC 3.0, especially in graphics and math. For example, Version 3.0 is 118% faster than Version 2.0 for circles. And with 8087/80287 math coprocessor support, the floating-point math is 160% faster than in Version 2.0.

New Microsoft QuickBASIC Language Extensions Make Programs Easier to Read

A number of new control structures such as SELECT CASE, DO WHILE, DO UNTIL, LOOP WHILE, LOOP UNTIL, and EXIT have been added to Microsoft QuickBASIC 3.0. These are statements similar to those found in Microsoft C and Microsoft Pascal. These statements, in addition to the block IF/THEN/ELSE/END IF (which was incorporated in Version 2.0), make structured programming easier.

The new named constants make your programs more readable and easier to maintain. Subprograms and multi-line functions have true local variables and allow you to call them by name and pass them parameters. These C-like structures let you write programs that are more well organized.

For more information on the products and features discussed in the Newsletter,
write to: Microsoft Languages Newsletter
16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717.
Or phone:
(800) 426-9400. In Washington State and Alaska, call (206) 882-8088. In Canada, call (416) 673-7638

Microsoft and the Microsoft logo are registered trademarks of Microsoft Corporation. ProKey is a trademark of RoseSoft, Inc. SideKick and SuperKey are registered trademarks of Borland International, Inc.

Latest DOS Versions:	
Microsoft C Compiler	4.00
Microsoft COBOL	2.20
Microsoft FORTRAN	4.00
Microsoft Macro Assembler	4.00
Microsoft Pascal	3.32
Microsoft QuickBASIC	3.00

Microsoft®

Ask Dr. Bob

The Driverless Printer

Dear Dr. Bob,
Could you tell me if it is possible in a Windows *application to send text and control characters to a printer for which there is no driver?—Wondering*

Dear Wondering,
If you're using a commercial *Windows* app, you're in trouble; you can't print without a driver. If you're writing your own app, you can't set up a DC (display context) and use standard GDI functions like Rectangle(), BitBlt(), and TextOut(). But you can use DOS file I/O to open, write, and close the DOS PRN: device. If you do this, it's up to you to understand the language your printer talks—its escape codes and special control characters. This, of course, ties your program to that specific printer and loses the device independence normally provided by *Windows*.

Math Library Functions

Dear Dr. Bob,
How can I use Microsoft C *math library functions in my assembler program?*
—Math Maven

Dear Maven,
Microsoft C, Version 4.0, has several floating-point options: in-line 8087 code, calls to an 8087 emulator, or calls to an alternative library. Before we tell you how to go about this task, we'll make a few disclaimers: C floating-point functions are meant for use by the compiler and handle arguments differently from standard functions, they're undocumented, and the fact that they work with Version 4.0 is no guarantee that they'll work with future versions.

To figure out how the compiler uses floating point, we used the /Fc compiler switch to examine the compiler's assembly language output and /FPa or /FPc so that the compiler would produce calls to the floating-point functions instead of 8087 code.

First, you have to initialize the library. If possible, let the regular C startup code handle this. If this won't work for you, look at the __CINIT function found in CRT0DAT.ASM from the C startup code disk. You should be able to pull the floating-point initialization code from this file.

To find the names of the floating-point functions and the arguments that each function requires, write small C programs

```
float a,b,c,d,e;
main()
 {a = (b+c)*(d+e); }

lea   bx,WORD PTR _b        ; put address of b into the bx register
call   __fldS               ; push b onto fp stack
lea   bx,WORD PTR _c
call   __faddS              ; add c to top of stack
lea   bx,WORD PTR _d
call   __fldS               ; push d onto fp stack
lea   bx,WORD PTR _e
call   __faddS              ; add e to top of stack
call   __fmul               ; multiply top two elements of stack
                            ; product replaces previous two
values on stack
lea   bx,WORD PTR _a
call   __fstsp              ; pop stack putting result in variable a
```

▲ **Figure 1** A sample C program and the assembler code it generates.

IN CODEVIEW, TO TRACE THROUGH A C SOURCE PROGRAM ONE MUST LINK WITH THE OBJECT MODULE. IT WOULD ALSO BE NICE TO HAVE CODEVIEW TRACE THROUGH THE C SOURCE PROGRAMS THAT HAVE MODULES WITHIN A LIBRARY. IS THIS POSSIBLE? A RELATED QUESTION: DOES THE EXEPACK LINKER OPTION REMOVE THE EXTRA DEBUGGING INFORMATION FROM SPECIFIED LIBRARIES? THIS WOULD AT LEAST ELIMINATE THE NEED TO COMPILE A FILE TWICE.

that demonstrate the floating-point functions you need and compile them with the /Fc switch. Study the assembler code and model yours after it. There are library functions for putting arguments onto a stack, doing math functions, and popping them off. **Figure 1** is a sample C program and the assembler code it generates.

You can explore other areas of compiler code this way, too. Another one you might want to look at is long int arithmetic.

You don't have to worry about formatting your numbers because MASM's DD, DQ, and DT directives use the IEEE formats for the real number constants that the library expects.

The floating-point functions documented in the C library manual, such as sin() and exp(), use standard C calling conventions, but before you use them, look at how the compiler converts all operands to double precision.

Using COM2 Without COM1

Dear Dr. Bob,
I just recently ran into a problem with QuickBASIC. *I opened COM2 for I/O and kept getting a bad filename error. The problem turned out to be that I did not have anything addressed as COM1 in my system. Everything worked fine after I installed a modem or serial card addressed as COM1.*

All that Microsoft product support would say is that you should not use COM2 if you do not have a COM1 already installed. I have designed IBM-compatible serial interfaces, so I know how the hardware works. To me they are two completely separate entities with two unique I/O addresses. Why am I having this problem, and why doesn't Microsoft consider it a problem? They told me they would not try to fix it. —*Constrained*

```
def seg _ &h0040
if (peek(0) + 256*peek (1))= &h02f8 then _
poke 2,&hf8: poke 3,&h02:_
print "COM2 patch made"else_
print "COM2 patch NOT made"
```

▲ **Figure 2** This program demonstrates how to patch the second word of RS232_BASE with COM2. You must run it before you run your *QuickBASIC* application that uses COM2.

Dear Constrained,
We're on your side. The hardware for COM2 can work without COM1. It seems that *QuickBASIC* looks in the BIOS data area at RS232_BASE. This area stores the addresses of up to four serial communications ports. When you start your system, the BIOS initialization code checks for serial ports, puts in addresses of any it finds, and sets the rest of the locations to zero. When both COM1 and COM2 are present, the address of COM1 (3F8h), is in the first word of RS232_BASE and the address of COM2 (2F8h) is in the second word. But when COM2 is the only port installed, its address, 2F8h, is in the first word, and the second word is zero. When *QuickBASIC* is loaded, it looks at RS232_BASE to see what serial ports are present, but it only recognizes COM2 when its address is in the second word.

Here's a way to work around this. The program shown in **Figure 2** will patch the second word of RS232_BASE with the address of COM2. You must run it before you run your *QuickBASIC* application that uses COM2. If you put this code at the beginning of your program, it will be executed too late to do any good because *QuickBASIC* checks for serial ports before it starts running your first BASIC statement.

C Queries and Suggestions

Dear Dr. Bob,
I'd like to offer some suggestions and ask some questions regarding Version 4.0 of Microsoft C. *In CodeView, in order to trace through a C source program, one must link with*

The Doctor Is In!

Dr. Bob wants to hear from you. Send him your questions and comments about *Windows,* MS-DOS, C, XENIX®, applications development, life after death, romance, war and peace, life on other planets, and life after applications development. Address them to Dr. Bob, c/o H. Roark & Associates, 141 E. 44th St., Suite 804, New York, NY 10017. Better yet, you can send your queries via MCI Mail. Our mailbox is called MSJ.

the object module. This is fine, but it would also be nice to have CodeView trace through the C source programs that have modules within a library. Is this possible?

A related question: does the EXEPACK linker option remove the extra debugging information from specified libraries? If so, this would at least eliminate the need to compile a file twice—once with the CodeView options and once without. This way I would compile everything with the CodeView options before creating the libraries. Then if I needed to debug a particular program from the library, I could link in the appropriate object file.

While using the watch option in CodeView, I tried to enter the following expression:

```
strptr[ (idx==0)
    ? 0
    : idx-1 ],s
```

*where strptr was declared as char *strptr[] and idx was an int. CodeView did not allow this and returned a message that ']' was missing. Am I missing something?*

Also, I often find myself setting several debugging options in CodeView (tracepoints, watchpoints, breakpoints, etc.), finding a bug, exiting CodeView, editing the source, compiling, linking, and then starting CodeView and having to reset the options I had before. It would be nice to have a save/reset option for CodeView that will save and later reset the tracepoints, watchpoints, and breakpoints. Here's what I suggest. I usually enter CodeView with the same options. There could be an environment variable (let's say CV) that would look something like "set CV= -w -f"so that these options will

```
printf ("starting available memory = %u\n",_memavl ());
if (NULL == (ptr=malloc ((unsigned) 5000)) )
    printf("Out of memory.");
printf("Available memory after malloc() = %u\n",_memavl () );
free(ptr);
printf("Available memory after free() = %u\n",_memavl () );
```

▲ **Figure 3** The last _memavl () does not reflect the amount of space now available from the prior free () call. This was done by using the small compiler model with the two options /Zi and /Od.

Figure 4: MAKE file template

```
# MAKE file template -  ###############################
# Assume the application program is called DOIT and the source
# directory is \CUSTOMER\SOURCE. The target directory for all
# obj and exe files is \CUSTOMER\S (which can easily be changed
# to \CUSTOMER\M or CUSTOMER\L by setting M=specific model
# type. The switches are:
#
#         M= model type S, M, or L
#
#          O= Compiler options (set for CodeView)
#
#          L= LINT_ARGS switch (currently set on)
#            To set this off, use L=
#
#          I= Switch to set environment variables
#
#            To set this off, use I=null
#
#          S= Switch to force a compile
#
#             To force a compile, use S=null
#
#########################################################
M=S
O=/Zi/Od
L=/DLINT_ARGS
D=\CUSTOMER\$(M)
I=null
S=\include\stdio.h

COMP=\bin\msc $*,$@ /A$(M) $(L) $(O);
LINK=\bin\link $(D)\doit.obj,$(D)\doit.exe,$(D)\doit.map,/M/E
LINKCV=$(LINK)/CO

# Make sure the include and library environ vars are correctly set
$(I):  \bin\inclib.bat
       \bin\inclib $(M)

$(D)\doit.obj:        doit.c $(S)
    $(COMP)

$(D)\doitcv.exe:       $(D)\doit.obj
    $(LINKCV)

$(D)\doit.exe:         $(D)\doitcv.exe
    $(LINK)
    dir $(D)\doit*.*
# End of MAKE file#####################################

The \BIN\INCLIB.BAT file looks something like this:

if "" == "%1" goto :default
set INCLUDE=\include;\gf\%1
set LIB=\gf\%1;\lib
goto :end
:default
set INCLUDE=\include;\gf\s
set LIB=\gf\s;\lib
goto :end
:end
```

```
/*
*  _maxavl() - Return the size of the largest block that
*              malloc() can allocate
*/
int _maxavl() {
  register unsigned size;
  register unsigned incr;
  char   *p;

/*
* Do binary search for largest block malloc() can return.
* Start search with a size of 32K. Keep going until the
* increment value is reduced to 0, with size homing in on
* its largest possible value.
*/
 for (size=incr=0x8000; incr; ) {
   incr >>= 1;   /* cut increment in half*/
   if ( (p=malloc(size))==NULL )
     size -= incr; /* malloc() failed, so decrease size */
   else {
     free (p); /* malloc()
succeeded, so free block */
     size += incr; /* and increase size */
      }
   }
/* If last malloc() failed, return size-1 instead of size: */
  return( (==NULL) ? size-1 : size );
   }
```

▲ **Figure 5** A simple function, _maxavl (), that returns the size of the largest block that malloc () can allocate.

CODEVIEW REEVALUATES THE ENTIRE EXPRESSION EACH TIME IT DISPLAYS THE WATCH WINDOWS. THIS MEANS THAT IF YOU'RE WATCHING AN ARRAY SUCH AS STPTR[CX], CODEVIEW MAY NOT ONLY WATCH A FIXED LOCATION; WHEN IDX CHANGES, YOU'LL BE WATCHING A DIFFERENT ELEMENT OF THE ARRAY.

now be the default for entering CodeView.

One more question, about malloc, free, and _memavl. If I have a program like the one shown in **Figure 3**, *the last _memavl () does not reflect the amount of space now available from the prior free () call. This was done by using the small compiler model with the two options /Zi and /Od.*

Finally, I would like to share some tips about the MAKE facility. I now set up MAKE files that look like **Figure 4**.

There might be standard include files in subdirectories \include,\gf\s, \gf\m, and gf\l, and standard libraries in subdirectories \lib, \gf\s, \gf\m, and \gf\l. — *C Buff*

Dear C Buff,

Yes, you are missing something: the CodeView manual, Section 4.4, which says CodeView supports a subset of C operators. The conditional operator ?: is not in the subset. That's the reason for the error.

One thing we found interesting about the complex operations CodeView does support is that it reevaluates the entire expression each time it displays the watch window. This means that if you're watching an array such as strptr[idx], CodeView may not only watch a fixed location; when idx changes, you'll be watching a different element of the array. This is not what we expected, but we like what CodeView does.

Your idea for the CodeView save/reset options is a good one. We'll pass it on. We, too, get tired of turning case sense off each time.

Your malloc(), free(), and _memavl() question is interesting. By marking the block as no longer in use, free() releases the memory. But as you discovered, this free space does not show up immediately in _memavl(). According to the C manual, _memavl() returns "approximately" how much memory is available. Because it doesn't add up the free blocks until they're needed, what it in fact returns is the largest size you can be sure you can allocate. You may, however, be able to allocate a much larger block, because if malloc() gets a request for a block larger than _memavl(), it goes down the list of free blocks joining adjacent ones, trying to make a large enough block. There is no way to force garbage collection analogous to the BASIC FRE($) function.

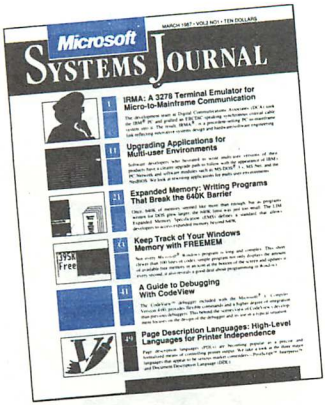
Figure 5 is a simple function, _maxavl(), that returns the size of the largest possible block. We suggest using it instead of _memavl(). ❑

Microsoft Corporation assumes no liability for any damages resulting from the use of the information contained herein.

Microsoft, the Microsoft logo, MS-DOS, and XENIX are registered trademarks and CodeView is a trademark of Microsoft Corporation. Macintosh is a trademark of Apple Computer, Inc. IBM is a registered trademark and Personal System/2 is a trademark of the International Business Machines Corporation. PageMaker is a registered trademark of Aldus Corporation. UNIX is a registered trademark of AT&T. Compaq is a registered trademark of Compaq Computer Corporation. Intel is a registered trademark of Intel Corporation. Paintbrush is a registered trademark of ZSoft Corporation.

Microsoft®

000-000-207


Microsoft
SYSTEMS JOURNAL
MARCH 1987 • VOL.2 NO.1 • TEN DOLLARS
IRMA: A 3278 Terminal Emulator for Micro-to-Mainframe Communication
Upgrading Applications for Multi-user Environments
Expanded Memory: Writing Programs That Break the 640K Barrier
Keep Track of Your Windows Memory with FREEMEM
A Guide to Debugging With CodeView
Page Description Languages: High-Level Languages for Printer Independence

SAVE $15.00 NOW

MICROSOFT SYSTEMS JOURNAL was created for you, the Professional Developer, to take you behind the scenes as the best minds in the industry tackle the toughest jobs.

Learn about the techniques, tools and strategies they use to build tomorrow's best-selling software products.

Get inside details about—

- New Operating Systems
- Mixed Model Programming
- Peer-to-Peer Communications
- Advanced Memory Management

YES! I want to be prepared for tomorrow's software innovations today! Enter my Charter Subscription at the special rate.

☐ 1 year (6 issues) for $34.95. ☐ 2 years (12 issues) for $64.95. ☐ 3 years (18 issues) for $89.95.

☐ Payment enclosed.
☐ Bill me.

Charge my:
☐ American Express
☐ Visa
☐ MasterCard

Name _____ Title _____
Organization _____
Address _____
City _____ State _____ Zip _____

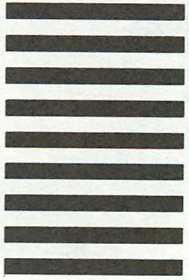
Card # _____ Exp. _____
Signature _____

Offer good in the U.S. and Canada only. Canadian subscribers add $10.00 per year to domestic rate. Full U.S. rate: $50/year. Offer ends September 30, 1987.

INTERNATIONAL SUBSCRIBERS PLEASE USE OTHER CARD .

E587A

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

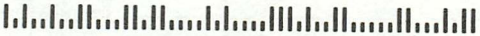


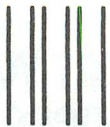
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 603 MARION, OH USA

POSTAGE WILL BE PAID BY ADDRESSEE

Microsoft Systems Journal
P.O. Box 1903
Marion, Ohio 43306



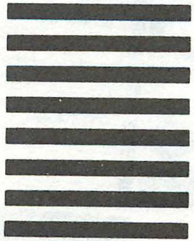


NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 603 MARION, OH USA

POSTAGE WILL BE PAID BY ADDRESSEE



Microsoft Systems Journal
P.O. Box 1903
Marion, Ohio 43306

MICROSOFT
SYSTEMS JOURNAL
FEBRUARY 1987 • VOL. 2 NO. 1 • TEN DOLLARS
Aldus: Preparing PageMaker for the Move to Windows
Moving Toward an Industry Standard for Developing TSRs
A Step-by-Step Guide to Building Your First Windows Application
New XENIX Version Will Be First to Run On the Intel 80386
A New Generation of Debugging Arrives with CodeView

GET YOUR OWN COPY OF MICROSOFT SYSTEMS JOURNAL—

the unique new bimonthly created for the Professional Developer!
Take advantage of the newest developments in hardware and software technology—the special insights, shortcuts, the dos and don'ts when working with these products.

Enter my subscription at special Charter Subscriber rate below:

☐ 1 year (6 issues) for $34.95.
☐ 2 years (12 issues) for $64.95.
☐ 3 years (18 issues) for $89.95.

Name_____ Title_____
Firm_____
Address_____
City_____ State_____ Zip_____
Tel:_____

E587B

☐ Payment enclosed.
☐ Bill me.
Charge my: ☐ American Express.
☐ MasterCard. ☐ Visa.

Card#_____ Exp._____
Signature_____

Offer good in U.S. and Canada only. Canadian subscribers add $10/year to domestic rate. Full U.S. rate $50/year. Offer ends September 30, 1987.

COUNTRY		1 YEAR RATE	2 YEAR RATE
Australia	*	A$65	A$120
Austria	*	Sch600	Sch1100
Belgium		BFr1800	BFr3250
Denmark		DKr325	DKr585
Eire		I£30	I£55
Finland		FMk200	FMk375
France		FFr275	FFr500
Italy	*	Lit60000	Lit110000
Japan	*	¥6000	¥11000
Netherlands		Fl100	Fl180
Norway		NKr300	NKr555
Portugal		Esc6500	Esc12000
Spain		Ptas6000	Ptas11000
Sweden		SKr300	SKr550
Switzerland	*	SFr65	SFr120
United Kingdom		£30	£55
West Germany	*	DM85	DM150
Other European	*	US$50	US$85
Other International	*	US$60	US$110

Orders accepted at these rates until September 30, 1987
*Return Postage Required

SPECIAL INTERNATIONAL RATES!

Direct international subscriptions are now available for *MICROSOFT SYSTEMS JOURNAL*, the unique new publication created especially for the professional software developer.

Stay up to date with the newest developments in hardware and software technology.

Get inside details about—

- Microsoft Operating Systems
- Peer-to-Peer Communications
- More Effective Debugging
- Using Expanded Memory

Enter my subscription at the special rate shown.

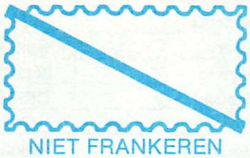
☐ Payment enclosed.
☐ Bill me.
Charge my:
☐ Access
☐ Visa
☐ American Express

Name _____ Title _____
Organization _____
Address _____
Country _____ Post Code _____

Card Number _____ Expiry Date _____
Signature _____ Date _____

U.S. AND CANADIAN SUBSCRIBERS PLEASE USE OTHER CARD

E587A



REPONSE PAYEE

PAYS-BAS

Microsoft Systems Journal
Int. Antwoordnummer
C.C.R.I. Numéro 454
2130 WB Hoofddorp, Holland

pcjs.org